

Models for Threat Assessment in Networks

Abstract

Central to computer security are detecting attacks against systems and managing computer systems to mitigate threats to the system. Attacks exploit vulnerabilities in the system such as a programming flaw. Threats are vulnerabilities which could lead to an attack under certain circumstances. The key to the detection of attacks is discovering an ongoing attack against the system. Mitigating threats involves a continuous assessment of the vulnerabilities in the system and of the risk these vulnerabilities pose with respects to a security policy. Intrusion detection systems (IDS) are programs which detect attacks. The goal is to issue alerts only when an actual attack occurs, but also to not miss any attacks. The biological immune system provides a compelling model on which to base an IDS. This work adds the biological concepts of positive selection and collaboration to artificial immune systems to achieve a better attack detection rate without unduly raising the false alarm rate.

Attack graphs assess the threat to the system by showing the composition of vulnerabilities in the system. The key issues with attack graphs are scalability to large networks, ease of coding new attacks into the model, incomplete network information, visualization of the graph and automatic analysis of the graph. This work presents an *abstract class model* that aggregates individual attacks into abstract classes. Through these abstractions, scalability is greatly increased and the codification of new attacks into the model is made easier when compared to the current approach that models each attack. Clustering of identical machines is used to reduce the visual complexity of the graph and also to increase scalability. Incomplete network information is handled

Report Documentation Page				Form Approved OMB No. 0704-0188	
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.					
1. REPORT DATE SEP 2006		2. REPORT TYPE		3. DATES COVERED 00-00-2006 to 00-00-2006	
4. TITLE AND SUBTITLE Models for Threat Assessment in Networks				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of California-Davis, Computer Science Department, One Shields Avenue, Davis, CA, 95616				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited					
13. SUPPLEMENTARY NOTES The original document contains color images.					
14. ABSTRACT					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES 176	19a. NAME OF RESPONSIBLE PERSON
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified			

by allowing “what if” evaluations where an administrator can hypothesize about the existence of certain vulnerabilities in the system and investigate their consequences. The patch management capability determines a mitigation strategy that optimizes the reduction of risk while maintaining a low cost. Unlike a vulnerability report, which only has a generic categorization of risk, this analysis uses the policy of the system to evaluate the risk. The resulting analysis is therefore linked to each system based on its system policy.

Models for Threat Assessment in Networks

By

MELISSA DANFORTH

B.S. (California State University, Bakersfield) 1999

M.S. (University of California, Davis) 2002

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Committee in charge

2006

Models for Threat Assessment in Networks

Copyright 2006
by
Melissa Danforth

Acknowledgments

The work was supported in part by a United States Department of Education Government Assistance in Areas of National Need (DOE-GAANN) grant #P200A980307. This work was also supported in part by ARDA under a subcontract from NetSquared, Inc. to UC Davis.

Contents

List of Figures	vii
List of Tables	ix
Abstract	x
1 Introduction	1
1.1 Detecting Attacks	2
1.2 Preventing Attacks	3
1.2.1 Attack Graphs	5
1.3 Major Contributions	8
2 Artificial Immune Systems	11
2.1 Introduction	11
2.2 Problem Description	13
2.2.1 The Problem of Novel Attacks	13
2.2.2 The Problem of Variants	13
2.3 Biological Models	17
2.3.1 Important Biological Features	17
2.3.2 Applying Biology to Security	20
2.4 Artificial Immune Systems	20
2.4.1 Affinity Maturation via Genetic Algorithms	22
2.4.2 Prior Work	23
2.5 Approach	25
2.5.1 Antibodies	26
2.5.2 Matching Function	26
2.5.3 Lifecycle	27
2.5.4 Genetic Algorithm Implementation	29
2.6 Implementation for Web Server Attacks	29
2.6.1 Data Set	31
2.6.2 Testing Parameters	32
2.6.3 Results	32
2.6.4 Notes on the Results	35
2.7 Conclusion	36

3	Attack Graphs: Overview of Concepts	37
3.1	Attack Graphs	41
3.1.1	Computation	41
3.1.2	Visualization	43
3.1.3	Uses of Attack Graphs	47
3.2	Related Work	49
3.3	Conclusion	51
4	Attack Graphs: Formalization and Mechanization	52
4.1	Introduction	52
4.2	JESS Overview	53
4.3	Formal Model	56
4.4	Method	57
4.5	Complexity	62
4.6	Exploit-Based Model	64
4.6.1	Experiments and Results	64
4.7	Conclusion	71
5	Abstract Class Model	72
5.1	Introduction	72
5.2	Motivation	74
5.3	Previous Works	77
5.4	Model	83
5.4.1	Preprocessing	91
5.4.2	Clustering	92
5.4.3	Postprocessing	93
5.5	Experiments	94
5.5.1	Comparison to Exploit Based Model	94
5.6	Conclusion	100
6	Analysis of Attack Graphs using Evolutionary Computation	101
6.1	Introduction	101
6.2	Related Works	102
6.3	Approach	104
6.4	Experiments	107
6.5	Conclusion	111
7	Future Work	113
7.1	Introduction	113
7.2	Artificial Immune Systems	114
7.2.1	Generalizing Antibodies	115
7.3	Attack Graphs	120
7.3.1	Computational Complexity	120
7.3.2	Visualization	120
7.3.3	Analysis	122
8	Conclusion	124

A	Artificial Immune System Framework	127
A.1	Program Description	127
A.1.1	Antibody Class	127
A.1.2	Webdata Class	128
A.1.3	AntibodyHeap Class	128
A.1.4	Lifecycle Program	128
A.1.5	Output Files	129
A.2	Tester File	129
B	Statecharts	130
B.1	Introduction	130
B.2	Statechart Features	130
B.2.1	Clustering and Refinement	131
B.2.2	Orthogonality	131
B.2.3	Entering States	132
B.2.4	Events and Actions	132
B.3	Syntax	133
B.3.1	Definitions	134
B.4	Semantics	135
B.5	Additional Features	138
B.5.1	Other Entrances	138
B.5.2	Timeouts and Delays	139
B.5.3	Overlapping States	139
B.6	Implementations	139
B.6.1	STATEMATE	139
B.6.2	Unified Modeling Language (UML)	140
B.7	Related Work	140
C	Exploit Based Attack Graph Rules	142
C.1	Rules from Published Papers	142
C.1.1	CMU Model	142
C.1.2	Sheyner Model	143
D	Abstract Attack Graph Rules	146
	Bibliography	159

List of Figures

2.1	An example range valued gene on the chromosome for an antibody, showing the value, offset pair.	26
2.2	Lifecycle of the antibodies	28
2.3	False positive trends across all population sizes for $r=0.7$ and $m=0.1$	33
2.4	False negative trends across all population sizes for $r=0.7$ and $m=0.1$	34
2.5	95% confidence intervals for the false positives and negatives when population size is 1000, $r=0.7$ and $m=0.1$	35
3.1	Tactical and strategic attack graph views for a five server network described. The highlighted path represents one path to get root on Host 3. The highlighted path is shown in Figure 3.2(b).	39
3.2	Hybrid attack graph view and the highlighted path for a five server network. The hybrid view shows all edges involving Host 2 as either a source or target machine.	40
3.3	Condition-oriented and exploit dependency based graphs for the highlighted path show in Figure 3.2(b).	44
3.4	Level by level breakdown of the attack sequence in the highlighted path of Figure 3.2(b). In level 1, two attacks, SSH from 0 to 1 and FTP from 0 to 3, are executed. In level 2, one attack, RSH from 1 to 3, is executed.	46
3.5	Level by level breakdown of the attack sequence in the highlighted path of Figure 3.2(b). In level 3, one attack, XTERM overflow on 3, is executed, giving the attacker root on Host 3.	47
4.1	Comparing the effects of the different LHS value test methods on the resulting Rete network. The inline comparison results in a more compact and efficient Rete network.	55
4.2	The affect of atomic attack set size, a , on the theoretical worst case number of edge labels and runtimes in relation to the number of machines in the network.	63
4.3	The three node sample network.	65
4.4	Attack graph for three node network presented in Sheyner <i>et al.</i> and Jha <i>et al.</i> . This shows the strategic and tactical graph views.	65
4.5	The two enclave sample network configuration	66
4.6	Strategic attack graphs for the four enclave scenarios. Graph (a) is scenario 1, (b) is scenario 2, (c) is scenario 3 and (d) is scenario 4. The exploits are remote to root (R2R), remote to user (R2U) and user to root (U2R).	67

4.7	The flat network topology.	69
4.8	Runtimes for the flat network topology.	70
5.1	The theoretical worst case runtime for an exploit based model versus an abstract model. The exploit model has 9 rules and 11 capabilities with an average of 6 patterns per rule. The abstract model has 7 rules and 9 capabilities with an average of 6 patterns per rule.	75
5.2	The effects of clustering on theoretical runtimes. The clustering is shown by its effectiveness in reducing the size of the machine set. In (a), the clustering is done on the exploit based model from Figure 5.1. In (b), the clustering is done on the abstract model from Figure 5.1 with the unclustered exploit based model runtime included for comparison.	76
5.3	The basic topology of the five subnet network used to compare the exploit based, abstract and abstract with clustering models. The DMZ subnet contains the machines that the attacker can directly connect to from his initial machine.	95
5.4	A comparison of the runtimes, the number of edges calculated and the number of initial variables for the non-abstract exploit-based model, the abstract model and the abstract model with clustering. Runtimes include preprocessing time.	97
5.5	Results for the clustering algorithm. In a), the maximum, minimum and average cluster sizes are shown. In b), the percent reduction in the number of hosts is shown.	98
5.6	Comparison of the visual complex of the attack graph using clustering and not using clustering. This is the attack graph for the network with 3 hosts per subnet.	99
6.1	Abstract model of the 3-node network presented in [1, 2, 3] with the initial conditions enumerated.	107
6.2	The effects of patching the <code>pe_local</code> vulnerability on Host 2. The patched node is dashed as are any nodes or edges disabled by this patch.	110
6.3	The effects of patching the <code>write_noauth</code> vulnerability on Host 2. The patched node is dashed as are any nodes or edges disabled by this patch.	110
6.4	The effects of patching the <code>pe_noauth</code> vulnerability on Host 1. The patched node is dashed as are any nodes or edges disabled by this patch.	111
7.1	An example of the coverage space of cross-reactive antigens.	116
7.2	The attack graph for a 56 host network that was clustered into 40 machine clusters.	121

List of Tables

2.1	Pseudocode for the selection of the children’s expressed attributes	29
2.2	Number of bits used (offset and value bit for range attributes) and valid values for all antibody attributes	30
4.1	The “main” and “edge” rules for the SSHD buffer overflow attack	61
4.2	Worst case complexity for Rete, JESS and the attack graph model. Rete complexity is from [4] and JESS complexity is from [5].	62
4.3	Enclave sample network: Connectivity for each scenario	68
4.4	Flat network: Segment A server names and vulnerabilities	69
5.1	Comparison of the RSH and IIS exploit based attack rules with the abstract attack rule.	73
5.2	The classification scheme for the PA study [6].	78
5.3	The classification schemes presented in Landwehr, et al.[7].	79
5.4	The classification schemes presented in Lindqvist and Jonsson [8].	81
5.5	The classification grammar presented in [9].	82
5.6	Classification scheme for the server, client, router and local entities.	85
5.7	Classification scheme for the attacker results.	86
5.8	Capabilities relating to what the attacker has gained.	86
5.9	Server related abstract rules in the abstract attack graph model.	87
5.10	Server capabilities in the abstract attack graph model.	88
5.11	Client related abstract rules in the abstract attack graph model.	89
5.12	Client capabilities in the abstract attack graph model.	89
5.13	Router related abstract rules in the abstract attack graph model.	90
5.14	Router capabilities in the abstract attack graph model.	90
5.15	Local binary related abstract rules in the abstract attack graph model.	91
5.16	Local capabilities in the abstract attack graph model.	91
6.1	The enumeration of all possible patches for the 3 node network in Figure 6.1.	109

Abstract

Central to computer security are detecting attacks against systems and managing computer systems to mitigate threats to the system. Attacks exploit vulnerabilities in the system such as a programming flaw. Threats are vulnerabilities which could lead to an attack under certain circumstances. The key to the detection of attacks is discovering an ongoing attack against the system. Mitigating threats involves a continuous assessment of the vulnerabilities in the system and of the risk these vulnerabilities pose with respects to a security policy.

Intrusion detection systems (IDS) are programs which detect attacks. The goal is to issue alerts only when an actual attack occurs, but also to not miss any attacks. The biological immune system provides a compelling model on which to base an IDS. This work adds the biological concepts of positive selection and collaboration to artificial immune systems to achieve a better attack detection rate without unduly raising the false alarm rate.

Attack graphs assess the threat to the system by showing the composition of vulnerabilities in the system: how the consequence of one attack can enable another attack. The key issues with attack graphs are scalability to large networks, ease of coding new attacks into the model, incomplete network information, visualization of the graph and automatic analysis of the graph. This work presents an *abstract class model* that aggregates individual attacks into abstract classes and generates abstractions. Through abstractions, scalability is greatly increased and the codification of new attacks into the model is made easier when compared to the current approach that models each attack. Clustering of identical machines is used to reduce the visual complexity of the graph and also to increase scalability.

Incomplete network information is handled by allowing “what if” evaluations where an administrator can hypothesize about the existence of certain vulnerabilities in the system and investigate their consequences. The patch management capability that the thesis introduces determines a mitigation strategy that optimizes the reduction of risk while maintaining a low cost. Unlike a vulnerability report, which only has a generic categorization of

risk, this analysis uses the policy of the system to evaluate the risk. The resulting analysis is therefore linked to each system based on its potentially unique system policy. Other types of analyses such as network design, forensics investigation and intrusion response are also supported by the general technique of this thesis.

Chapter 1

Introduction

Computer security is a multi-faceted field that covers a broad range of topics. This work primarily addresses the problems of detecting attacks against computer systems and managing computer systems to prevent attacks. A computer system can range from an individual machine to a network of machines. Each system has a policy that defines the expected behavior of the system. This policy can be either explicitly stated or implied by the system's context. Typically a policy contains the system's requirements with respect to confidentiality, integrity and availability. For example, a policy might declare the services provided by the system or the privileges granted to specific entities within the system.

An attack on a computer system is an attempt to violate the policy of the system, usually to achieve an attacker's goal. Attacks use weaknesses in the system to violate the policy. These weaknesses are called *vulnerabilities* of the system. The motivation for attacks and the goals of attacks are varied. Some attackers are motivated by a quest for fame, others by fortune, and still others have additional motivations such as to be a nuisance. The goals of an attack can range from acquisition of resources or data to simply seeing what the consequences of an attack are. Attacks can be highly automated – only requiring a person to launch the attack. Attacks can also be done step by step by an attacker or can trick an innocent third party into executing steps of the attack. Attacks can be related to a previously seen attack, using the same basic code with some modifications. Attacks can also be new, using a new code base or targeting a new vulnerability.

1.1 Detecting Attacks

The problem of detecting attacks falls to systems called intrusion detection systems (IDS). The goal of an IDS is to detect, ideally, all attacks launched against the system that the IDS is protecting. When an IDS fails to detect an attack, this is called a false negative since it labels the data corresponding to the attack “negative” for an attack when, in fact, an attack occurred. Related to the detection of attacks, the IDS should also only alert when an actual attack is observed. To alert on innocuous data is called a false positive since the system has reported an attack when none occurred. Novel attacks and new variants of prior attacks are more prone to being labeled false negative, particularly with detection algorithms that are optimized to known attacks. Certain algorithms are also more prone to false positives. Ideally, an IDS algorithm would have both low false positive rates and low false negative rates.

The three most common types of IDS algorithms are anomaly [10, 11], misuse [10, 11] and specification [12, 13, 14] algorithms. Anomaly algorithms use statistical techniques to develop a profile of “normal” or expected behavior. This allows them to detect novel attacks and new variants that fall outside the realm of “normal” behavior and thus achieving a lower false negative rate. However, since anomaly algorithms typically use statistical techniques, outliers in innocuous data may fall out of the statistical range of “normal”, causing an increase in false positives. Misuse algorithms typically use a database of unique signatures of known attacks. Any repeat instance of the known attack will be detected by the signatures. A well-crafted signature will not alert on “normal” behavior, so the false positive rate is low with misuse algorithms. However, signatures are ill-suited to detect novel attacks or new variants, so the false negative rate is high. Specification algorithms use the specification of the system to define all possible “normal” behavior. In other words, the specification algorithm will model all possible valid behaviors. Any behavior which falls outside of the model of “normal” is considered an attack. The false negative rate will be low. As long as the specification is complete, a specification algorithm will have no false positives. However, it is difficult to fully specify all “normal” states in a system. Any “normal” behavior that is not captured by the model may be reported by the IDS as a false

positive as the IDS is not aware that this is “normal” behavior.

The biological immune system is a tempting model on which to base an IDS algorithm. The immune system must also have a low false positive rate and a low false negative rate. A false positive in a biological immune system induces an auto-immune response, where the body attacks itself. A false negative in a biological immune system leads to an unchecked infection which can be fatal to the organism. Biological immune systems achieve this balance between false positives and false negatives through several processes: negative selection, positive selection, collaboration, affinity maturation and peripheral tolerance. These processes are detailed in Section 2.3.

Artificial immune systems (AIS) use concepts from biological immune systems to create an IDS algorithm that have low false positive rates. However, most AIS algorithms proposed to date [15, 16, 17, 18, 19, 20] focus on negative selection, so that their false negative rate may be too high for certain problem domains. This work adds the concepts of positive selection and collaboration to achieve a much lower false negative rate without unduly sacrificing the low false positive rate. Tunable parameters are used to select the level of positive selection and collaboration so that an administrator can select the level that yields acceptable false positive rates and false negative rates for the system.

1.2 Preventing Attacks

Beyond detecting that attacks have occurred, another facet of computer security is prevention on the part of system administrators. Attack prevention includes assessing the risk the computer system faces, determining the level of acceptable risk with respects to the policy of the system and mitigating the unacceptable risks. Risk assessment is the critical first step because any risks that are missed could negate the effect of the mitigation strategy. To use an analogy to physical security, if one overlooks the basement windows while assessing the risk to one’s house, it does not matter how many alarms are put on the doors and upstairs windows, the intruder can still enter undetected through the basement windows. The level of acceptable risk is determined both by security requirements of the policy and the costs associated with certain mitigation strategies. To return to the physical

security example, the acceptable risk to an average homeowner is likely much different than the acceptable risk to a top secret government facility. Additionally, alarm systems which are too costly for the homeowner may be a perfectly acceptable cost for the government facility. The mitigation strategy is a list of procedures to execute to reduce or eliminate unacceptable risks. A costs versus benefits analysis is often performed to determine the list of mitigation procedures.

Traditionally, risk assessment has been done in part with software called vulnerability scanners which would scan the system in search of known vulnerabilities. Vulnerability scanners, however, have several shortcomings. They can only detect testable vulnerabilities on a system. If a vulnerability is untestable due to the environment of the system or the nature of the vulnerability, then the vulnerability scanner will not detect it. For example, an environmental factor in testability is the presence of a firewall that may block the connection request by the vulnerability scanner. If the vulnerability is only testable by a process on the system and the scanner is remote, it will also not be able to detect the present of the vulnerability.

Vulnerability scanners also only report vulnerabilities detected. There is no information about how vulnerabilities may interact within the system. A common scenario is the “foothold” situation. In this scenario, there are several machines protected from the public by a firewall (“private” machines) but they communicate with one machine that is accessible to the public (the “public” machine). An attacker uses a vulnerability to gain a “foothold” on the public machine. He then uses the public machine as a launching pad to attack the private machines over the communication channel between the public and private machines. A vulnerability scanner would report the vulnerabilities on the public and private machines, but it would not show *how* the public machine could be used as a “foothold” to attack the private machines.

Additionally, the “threat level” assigned to various vulnerabilities by a vulnerability scanner is static. It is based solely on how that vulnerability can affect the single machine on which it is found. A more useful metric would be to consider the policy of the system to see if the effect is significant and to consider how that vulnerability enables foothold situations within the system. This would allow the administrator to determine risk

based on the nature of his system. Such a dynamic, situationally dependent assessment of risk is not possible with basic vulnerability scanners.

1.2.1 Attack Graphs

Attack graphs are a method by which one can explore the interactions between vulnerabilities across the whole system. For purposes of representation as an attack graph, a system is a network of machines. Attack graphs can take the information provided by a vulnerability scanner and analyze it further to determine the interactions between vulnerabilities, such as to reveal the presence of “foothold” situations. Attack graphs model an attack as a set of preconditions required for the attack and a set of postconditions enabled by the execution of the attack [21]. The postconditions from one attack may be a subset of the preconditions for other attacks, enabling other attacks by satisfying their preconditions. Thus, attack graphs show how an attacker might fully penetrate a network given the interaction between one attack’s postcondition and another attack’s precondition. For example, with the public machine and private machines “foothold” scenario, a precondition of the attacks against the private machines is that there is a communication channel between the private machine and an attacker controlled machine. When the attacker compromises the public machine, he gains access to the communication channel between it and the private machines. This satisfies one of the preconditions of attacking the private machines, so he can now launch attacks against the private machines.

The main issues within the attack graph domain is the automatic generation, visualization and automated analysis of attack graphs. Automatic generation encompasses several subissues: scalability to large networks, ease of coding new attack rules and gathering network information. In order to be scalable to large networks, the algorithm to calculate the attack graph needs to run in reasonable time and not exhaust the available memory on the average workstation used by administrators. Many of the prior works in the attack graph field [1, 2, 3, 22, 23, 24, 25, 26, 27, 28] are not scalable to larger networks. The language used to define the attack model should also be flexible enough to easily allow the addition of new rules without extensive hand coding. Several of the prior methods [1, 2, 3, 22, 23, 24, 25, 26, 27, 28] require that the details of each new attack be enumerated

even if that attack closely resembles another attack already present in the system. Gathering information about the network is another issue with automatic generation. The attack graph model needs to capture the initial vulnerabilities present in the network in order to compose them into an attack graph. As mentioned above, vulnerability scanners may not have complete information about the network. The data gathering phase needs to be flexible enough to allow an administrator to fill in information or to perform “what if” scenarios to see what would happen if that vulnerability were present.

Visualization is another major issue with attack graphs. Often, the graphs contain so many paths that it is difficult for a human analyst to see the details of each path. Many of the prior works either just presented this visually complex graph [1, 2, 3, 22, 23, 24, 25, 26, 28] or methodically aggregated nodes to reduce visual complexity [27]. The prior works did not consider visualization in context of the policy of the network. Even the aggregation method used a static hierarchy to determine aggregation rather than aggregate with respects to the policy. Visualization could be tied to the automatic analysis to display the paths that the analysis deems crucial while aggregating or hiding other paths.

The automatic analysis of attack graphs depends on the use of the attack graph. If the attack graph is being used for risk assessment, the analysis would state the risk in terms of how the attacker has violated the policy of the system. For example, if the policy said that unauthorized users should not be able to get user access on host xyz, the risk assessment would say if this policy has been violated. This is a very simple analysis of the attack graph that involves finding the presence of nodes in the graph that violate the policy. The prior works [1, 2, 3, 22, 23, 24, 25, 26, 27, 28] can all do this sort of analysis. More interesting uses for attack graphs are patch management and network design. For these uses, the analysis would return a mitigation strategy. The prior works [1, 2, 3, 22, 23, 24, 25, 26, 27, 28] do not consider the policy when determining the mitigation strategy, although some consider a cost metric associated with each measure. In order to be useful to administrators and to automate the cost versus benefits analysis, the site policy must be considered when deriving the mitigation strategy. Additionally, the prior works [1, 2, 3, 22, 23, 24, 25, 26, 27, 28] have focused on patching initial vulnerabilities as the method used in the mitigation strategy. There are several other techniques that can be used to mitigate risk such as adding firewall

rules to block connectivity or placing IDS sensors to detect attack activity.

This work addresses these five issues with attack graphs: scalability, ease of coding new attacks, incomplete network information, visualization and mitigation strategy analysis. More scalability over prior works is achieved by using an expert system instead of a model checker. A model checker requires the enumeration of all possible states while the expert system can search the database looking for matches to preconditions of an attack. This work further extends the scalability by clustering identical machines and creating an abstract attack model. Clustering identical machines reduces the number of machines that need to be evaluated. The abstract attack model classifies attacks by common preconditions and postconditions. If two attacks differ only by the name of the vulnerability in the preconditions, they would be abstracted into one abstract class. The abstract attack model increases scalability by reducing the number of rules required to model the system. The performance of expert systems is increased when the number of rules are decreased.

The abstract model also makes it easier to code new attacks into the model. If the new attack falls under an existing class, it can be added just by adding the attack's information to the algorithm that imports vulnerability information into the abstract model. Typically, this involves just adding the name of the vulnerability for that attack to the list of vulnerabilities covered by the abstract class. Additionally, this work uses a language for the rules that is supported by several expert systems. This makes the rules portable to a variety of expert systems without requiring recoding. For handling incomplete network information, "what if" scenarios can be run to investigate the consequences of unknown vulnerabilities being present on the system.

The visualization of attack graphs is kept separate from the method of computation. This allows optimal methods for each to be selected. The visualization method supports several views of the data. The thesis introduces three ways to project attack graphs, depending on the information needs of the human analyst. The *tactical* view shows all possible paths in the attack graph. The *strategic* view is a "there exists" view of the attack graph, which shows the existence of exploit paths to compromise the critical resources as indicated in the policy, but not all possible paths. The strategic view is less visually complex than the tactical view. A *hybrid* view allows the administrator to have strategic

view of most of the attack graph, but expand certain portions to the tactical view. Additionally, methods for aggregating nodes with respects to the policy would further reduce visual complexity.

The mitigation strategy analysis uses evolutionary computation to find hardening measures that have low cost and high benefit. The cost and benefit of a hardening measure is determined by the policy and the type of analysis. For example, if the policy states that a service must be publicly accessible, then the cost of the hardening measure “block access to service” would be high. The type of analysis affects the weight given to each type of hardening measure. When analyzing a network as part of the network design phase, the class of hardening measures relating to connectivity between machines would be more beneficial than the class relating to patching vulnerabilities. However, the opposite is true when analyzing the attack graph as part of patch management. While the prior works consider just patch management analysis, this work also considers network design, forensics and intrusion response. For forensics, the analysis finds the likely exploit paths the attacker has used based on the evidence and shows what further paths the attacker would have been able to take. This provides direction in determining how far the attacker has penetrated the network. A similar analysis is performed for intrusion response, but it also determines mitigation strategies to prevent the attacker from progressing further.

1.3 Major Contributions

The artificial immune system work adds the concepts of positive selection and collaboration. This decreases the false negative rate, allowing more attacks to be detected, without increasing the false positive. Through this system, an administrator can dynamically adjust the settings for positive selection and collaboration to balance the sensitivity to possible attacks with the rate of false alarms.

The attack graph work addresses five issues with attack graphs: scalability, ease of coding new attacks into the model, incomplete network information, visualization and mitigation strategy analysis. Scalability is increased by using an expert system to compute the attack graph rather than a model checker which requires the enumeration of all possible

states. Scalability is also greatly increased by clustering identical machines and creating an abstract class model. Coding new attacks into the model is also made easier by the abstract class model. A new attack that falls under an existing abstract category can be added just by modifying the model to specify what category the attack belongs to.

Incomplete network information is handled by allowing “what if” scenarios that can hypothesize about the existence of vulnerabilities and investigate the consequences. Visualization complexity is reduced by the abstract model and clustering as both methods reduce the number of nodes and edges in the resulting attack graph. Three visualization views are also supported to allow the human analyst to select a desired level of visual complexity. The tactical view is the most complex as it shows all possible paths in the attack graph. The strategic view is less visually complex as it just shows that there exists a path to compromise each node instead of all possible paths. A hybrid view allows the analyst to have a strategic view for most of the graph, but he can expand any portion he selects to the tactical view.

The mitigation strategy analysis adds the consideration of the security policy while determining the strategy. Prior works did not consider the policy of the network. If two identical networks with different policies were analyzed with the prior works, the same strategy would be derived. This work incorporates the policy in the analysis so that such a scenario could result in two different strategies depending on the nature of the policies. Additionally, the prior works only considered patching machines while this work also considers placing firewall rules to block connectivity and adding rules to the intrusion detection system to detect attacks. This work also supports multiple types of analysis: patch management, network design, forensics investigation and intrusion response.

Chapter 2 covers the artificial immune system work. It details the problems with novel attacks and attack variants, the important features of biological immune systems and the implementation of the artificial immune system for detecting web server attacks. Chapter 3 introduces the concepts relating to attack graphs such as computation, visualization and uses. Chapter 4 goes into further details about attack graphs. It justifies the selection of an expert system for computation, presents a formal model of attack graphs and gives the exploit-based implementation of attack graphs. Chapter 5 details the abstract class model

and clustering for attack graphs. Chapter 6 gives the evolutionary computation method of analyzing attack graphs to develop a mitigation strategy. Chapter 7 details future work in both the artificial immune system and attack graph areas.

Chapter 2

Artificial Immune Systems

2.1 Introduction

Currently, most intrusion detection systems (IDS) can be classified as anomaly [10, 11], misuse [10, 11] or specification [12, 13] based. Anomaly based IDS profiles the normal activity of the system and monitors for abnormalities. Misuse based IDS typically uses signatures of known attacks and monitors for matching events. Specification based IDS make use of specifications of the allowed uses and behaviors of the system or programs and watches for deviations. Each of these approaches has its strengths and weaknesses.

Anomaly systems are typically based on statistical methods, rules or immunological concepts [10, 13]. Anomaly based IDS can detect novel attacks or new variants of attacks in most cases, but it can also suffer from high false positives. Since the anomaly systems generate alerts for anything that does not fit the normal profile, even new normal behavior will generate an alert. There are also several published techniques to bypass anomaly systems such as injecting attack traffic or parts of the attack traffic into the normal training data in order to bias the learning [29]. Another technique is to carefully craft the attack such that it falls within the normal coverage space of the IDS [29].

Most misuse detection systems are typically called signature IDS. Signature based IDS cannot detect most new attacks unless they closely resemble one of the known attacks. This kind of IDS also has trouble detecting variants of attacks unless the attack can be easily generalized. Well crafted signatures can reduce false positives, but poorly crafted

signatures can cause excessive false positives.

Specification IDS is one solution to the problem of anomaly systems generating too many false positives. The idea is that by building a specification of all allowed behaviors, any normal behavior, even unseen normal behavior, will not generate an alert. However, specification IDS has its own set of problems. Primarily, there is the problem of the difficulty of specifying the complete allowed behavior of the monitored system. Beyond the difficulty of creating the specification, there is the overhead of comparing abstract activity against what could be a complex specification. There is also the maintenance of the specifications as the system is changed over time. Several papers have been published on the development of specifications [14, 13, 12, 30]. In particular, [14] compares the issues with specification IDS to the issues with anomaly and misuse IDS.

As seen from the above overview, some issues in IDS are false positive rates, detecting novel attacks, detecting variants of attacks and the setup and operational costs associated with the system. Another issue that is gaining more attention in recent years is the management of alerts sent to users, particularly creating alerts that have useful and actionable information for the users. The biological immune system and machine learning methods provide some inspiration to overcome some of these issues. Most prior immunological systems [15, 16, 17, 18, 19, 20] focus on the concept of negative selection but ignore other key concepts from immunology. Prior uses of machine learning methods have also focused mainly on anomaly detection [10] or using genetic algorithms within negative selection algorithms [15].

Section 2.2 describes the issues with novel attacks and attack variants. In Section 2.3, important concepts from the biological immune system are detailed. This section also outlines how the biological concepts relate to computer security. Section 2.4 introduces the artificial immune system for detecting attacks. Section 2.5 details the artificial immune system used in this thesis. In Section 2.6, the implementation of the AIS for detecting webserver attacks is detailed and experimental results are presented.

2.2 Problem Description

2.2.1 The Problem of Novel Attacks

Many new vulnerabilities are discovered in various systems and programs every week. This creates a wealth of possibilities for people to develop new attacks. Passage of time from vulnerability discover does not seem to affect the development of a new attack. Attacks can be released into the wild months after the vulnerability was released. Usually these attacks are very successful because many systems are not up to date with patches. There is also the rarer zero-day attack which exploits a previously unreleased vulnerability.

The biological immune system had been a desirable model for detecting new attacks since it defends the body against a variety of unknown attacks. However, as mentioned previously, most systems only employ a subset of features of the biological immune system. This is discussed more in depth in Sections 2.3 and 2.4.

2.2.2 The Problem of Variants

Attacks can also have various forms. These variants can be created either by other attackers or automatically by the attack itself. Polymorphic and metamorphic worms are historic examples of attacks with automatic variants. Usually the man-made variants take days to appear as people modify the original attack that is circulating in the wild. Automatic variants have been rare, but are now commonplace and can be produced far more rapidly since the original attack carries its own variant generator engine.

Man-Made Variants

Many times, the presence of a new attack on the Internet creates many variants as other people take the attack and modify it. These variants exploit the same vulnerability and often use the original attack as a basic template. Thus there are usually several similarities in the code. Various examples of such man-made variations can be found looking through various antivirus vendors' virus information libraries. Variants that have grabbed the headlines in recent years include the variants of the CodeRed and SoBig [31] attacks.

Code Red for example first appeared on the Internet around July 12, 2001. It

exploited a vulnerability that had been announced the month before by eEye Digital Security [32]. On July 17, 2001, the team at eEye published a detailed analysis of the worm which they termed “Code Red” after the Chinese references in the website defacement and the soft drink, Mountain Dew Code Red, that they were drinking while disassembling the code. The main purpose of the worm was to launch a denial of service attack on www.whitehouse.gov. It also defaced the websites of US English Windows NT or 2000 servers. The eEye team highlighted one main problem with the spread of this worm. When spreading to other machines, it used a static seed in its random IP generator that caused all versions of the worm to generate the same list of IPs. It also targeted the denial of service attack by the IP of www.whitehouse.gov instead of using domain name resolution [33]. Two days later, reports came of a new variant of this worm which used a random seed to compute the IP of the machine to infect. This is a prime example of a person taking an existing worm and modifying it, in this case to fix a bug in the original worm’s code.

A month later on August 4, 2001 eEye released an analysis [34] of a new worm that exploited the same vulnerability as Code Red. This worm was dubbed CodeRed II because that string was contained within its code, but eEye pointed out it has a completely different payload than Code Red and thus is not technically a variant in their eyes. The random IP generator for spreading masks the results such that 1/2th the IPs will be the same class A as the local host, 3/8th will be the same class B and the remaining 1/8th will be outside those IP ranges. It also regenerates the IP if it is the same as the local host, 224.x.x.x or local loopback. The worm also only spreads if the date is before October and not 2002. Otherwise it reboots, effectively removing the spreading portion of the worm unless it is reinfected by another system with an improper date. This essentially limits the spread to before October 2001.

CodeRed II also installs multiple backdoors that allow the attacker to execute arbitrary commands on the server. The first backdoor copies %System%\cmd.exe to two common IIS script directories on the c:\ and d:\ drives. Most IIS installs use the default paths on either the c:\ or d:\ drive, so the infection stage would usually succeed. The worm also changes the permissions on the script directories so that a simple request such as:

`http://infected_machine/scripts/root.exe?/c+dir`

from a web browser would execute the `dir` command and thus use the backdoor.

The second backdoor is a trojaned `[d—c]:\explorer.exe` binary. The `explorer.exe` file is the default shell of most Windows users. It is loaded every time a user logs into the machine. Since the Windows search path usually has the `c:\` or `d:\` directory before the normal `explorer.exe` directory, the trojan `explorer.exe` binary will be executed instead of the normal one. The trojan first executes the normal `explorer.exe` so that the user does not see anything unusual. Then the trojan goes into an infinite loop which first disables the file protection and then changes the permissions on the script directories and then creates two virtual web directories which map to the `c:\` and `d:\` drives. The new virtual directories are given the same permissions as the script directories. Hence, the request:

`http://infected_machine/c/winnt/system32/cmd.exe?/c+dir`

would execute the `dir` command as with the first backdoor. By disabling the file protection, the attacker could also send a command to remove or change system files that would normally be protected.

These backdoors were subsequently used by other worms such as Nimda [35]. And in a sign that even the long passage of time does not deter people from creating variants, a new variant of CodeRed II called CodeRed.F by most vendors was discovered March 11, 2003. This variant changed the year portion of the CodeRed II worm from 2002 to 34951, but leaves the month check [36]. Thus the worm will likely spread until October of 2003, then fade out as the month check causes the infected systems to reboot.

Automatic Variants

Worms are one type of attack which are likely to use automatic methods to create variants of itself. Often these worms are classified collectively as polymorphic or metamorphic worms, although the methods used can vary. These methods can be divided into the following categories, although only the last two categories fall under a strict definition of polymorphic or metamorphic worms.

disguise Change some facet of the delivery vector so that it appears different. For example, email worms may have a random subject line, message body or attachment filename.

Intended to trick users and make it harder to do basic filtering.

cross-platform Can infect multiple operating systems and/or architectures.

multi-vector Uses multiple vectors of infection, such as exploiting multiple vulnerabilities.

modularity Uses plug-ins, roles and other forms of modularity so various worms have different functionalities.

mutation Use equivalences in machine code or programming logic to create code that looks different but has identical function. The worms also may add "junk" code. In shell code, this junk is often NOPs or equivalent instructions. In those worms that recompile themselves from source or use scripting languages, the junk may be comments or "do nothing" functions. The worms may also use out-of-order code, utilizing jumps, to prevent simple string comparisons on sequences of instructions.

encryption Encrypt the shell code or worm payload. Requires a decryption engine. Simple encrypted worms use a static decryption engine that makes detection easy. Complex encrypted worms can use mutations on the decryption engine.

Recent examples of worms that use these methods are the Windows targeted worms NewLove [37], Hybris [38, 39] and Magistr [40]. NewLove uses a simple mutation: inserting random comments into its scripting code. Hybris uses modularity in the form of plug-ins and simple encryption of the plug-ins. Some of its plug-ins also use encryption while infecting files. Magistr is a more complex polymorphic email worm that uses disguise, encryption and mutation. Disguise, in the form of randomly constructed subjects and message bodies, is used when propagating itself by email to other machines. Encryption and mutation are used when infecting a file. An infected file has the entry point overwritten with "junk" code that eventually jumps to the end of the file. At the end of the file is the encrypted body of the worm. It also varies its behavior when infecting a machine. It probabilistically selects one of two methods (a variant introduced a third method) to cause itself to be loaded into memory at startup.

2.3 Biological Models

The biological immune system provides many key features that can be used for the design of an effective intrusion detection system. The purpose of an immune system is to prevent pathogens from invading the organism by detecting and eliminating the pathogen. Pathogens contain certain structures on their surface by which they can be recognized by the immune system. The structures which trigger an immune response are called antigens. The ability of a biological immune system to detect previously unknown pathogens makes it an appealing inspiration and model for an intrusion detection system.

2.3.1 Important Biological Features

Multiple methods of defense from pathogens is one key feature of immune systems. The biological immune system is a combination of barriers, innate responses and adaptive responses. The most obvious barrier in a living creature is the skin. Other barriers include mucus membranes in the lungs and hydrochloric acid in the stomach. The primary purpose of barriers is to prevent pathogens from entering the body. An analogy in the computer arena would be a firewall at the perimeter of a network. The innate responses include certain cells called phagocytes which seek out and engulf pathogens. Certain phagocytes have pattern recognition receptors which are generic patterns that allow the phagocytes to recognize common pathogens. The adaptive response is the system that often comes to mind when the immune system is mentioned. The adaptive system involves cells called lymphocytes which can react to more specific antigens. The two most common lymphocytes are T cells and B cells. Both these cells randomly generate receptors to recognize antigens that do not correspond to self cells, but they differ in their functionality. B cells typically function to find antigens and generate antibodies which trigger a response. T cells consist of several types of cells. Helper T cells are needed to activate other immune system cells. Cytotoxic T cells destroy pathogens. Suppressor T cells suppress the activity of other immune system cells.

The interaction between cells is another key feature of the biological immune system. There can be either independent or collaboratory responses to pathogens. For example,

phagocytes can find and destroy pathogens which match their pattern recognition receptors independently from the rest of the immune system cells. On the other hand, most of the immune responses are actually a complex series of interactions between the various parts of the immune system. The basic stages of a reaction is finding an antigen, binding to it and activating a response which eliminates the pathogen. In this scenario, it is the B cells activation which creates the antibodies that direct the rest of the immune system to eliminate the pathogen. However, when a B cell finds an antigen to which it can bind, most of the time it will not activate without a signal from an activated helper T cell. A helper T cell will only become activated when it finds a self protein called MHC that is bound to a foreign antigen. This complex of MHC and foreign antigens can occur through several processes, such as when a phagocyte engulfs and destroys a pathogen. A cell which has the complex of MHC and the foreign antigen is called an antigen presenting cell since it presents the antigen to T cells. Thus a feedback loop involving other components of the immune system that have detected the antigen must be in existence most of the time for B cells to activate.

Another feature of B cell activation that is interesting to look at is affinity maturation. Affinity is the measurement of how well an antibody binds to an antigen. The more an antibody binds to an antigen, i.e. higher affinity, the less likely it is to become unbound from the antigen before a response cell can eliminate the antigen. Affinity maturation is the process by which antibodies gain more affinity for an antigen. This occurs during the process of responding to an antigen. When a B cell is activated, it creates copies of itself which become either plasma cells or memory cells. Plasma cells are responsible for generating antibodies while memory cells allow the immune system to respond to the antigen more rapidly, especially in the future. During this cloning and differentiation of the B cell, mutations can occur. When these mutations result in B cells, and thus antibodies, which have a higher affinity to the antigen, these higher affinity cells outcompete the lower affinity cells for binding to the pathogen. Thus, the higher affinity cells become more prolific. This process may occur over many cycles and is called affinity maturation.

As mentioned previously, another feature of the immune response is the differentiation of activated B cells into memory cells. Due to affinity maturation, these memory cells

usually have higher affinity to the antigen than the first B cell which detected the antigen. These higher affinity memory cells trigger a more aggressive response to the antigen in the future, usually preventing reinfection by the pathogen. The creation of these memory cells is the basic idea behind vaccinations.

One key feature of the biological immune system that is used heavily in certain artificial immune system applications is negative selection. Negative selection is a process B and T cells go through while they are maturing. New B and T cells create their antigen receptors randomly. This leads to a high probability that a receptor which matches self will be generated. Such a receptor would be bad as it could lead to the immune system attacking the organism's own cells. One method to prevent this is negative selection. During maturation of the cells, they are presented with a set of self antigens. If they react to the self antigens, they are deactivated or destroyed. One exception to this is during the generation of helper T cells. Since helper T cells recognize foreign antigens bound to self MHC, those helper T cells with a low affinity to self MHC are allowed to survive. This exception is called positive selection.

Another method by which the immune system avoids reacting to self is called peripheral tolerance. This is essentially the deactivation of cells after maturation which react to self. Such cells could be created during the mutation of the B cells when it clones itself after being activated for example. Also, cells that are self-reactive may pass through the maturation process without being destroyed if there is not a sample of that self antigen in the tissue where the maturation occurs. One method to get rid of these self-reactive mature cells is anergy, which occurs when the collaboratory signals required for B and T cell activation never occur. Another method is the presence of cytotoxic T cells which can detect and destroy self-reactive B cells.

More information about the biological immune system can be read in [41, 42, 43]. Basic overviews that go into some detail are contained in [41, 43] while [42] goes into more detailed about the biological, genetic and chemical processes.

2.3.2 Applying Biology to Security

The most common application of biological systems to security seen to date is the use of artificial immune systems (AIS) to detect attacks both for network intrusion detection and host intrusion detection. Most of these systems are somewhat limited, however, in that they do not take into consideration the collaboratory portion of the immune system nor the multiple methods of detection. One particular aspect of interest that has not been investigated fully is peripheral tolerance. This could be accomplished by a variety of methods. One method could be an interactive system that allows a user to deactivate an antibody that is causing too many false positives. Another method could be to have a system akin to the cytotoxic T cells which could be developed using positive selection.

Interoperability between systems is another key biological concept. Given the current state of technology, it seems unlikely one single system will be able to detect all sorts of attacks. Rather, a layered approach inspired by the biological immune system shows more promise. There could be peripheral barriers to entry that are analogous to skin and mucus membranes which would act as a crude method to prevent attacks. An example of such would a firewall which prevents traffic from certain ports from entering the network. Within the network there could be distributed systems which could act independently or collaboratively to detect attacks. These systems could be adaptive like the AIS described in section 2.4 or a currently available signature based system such as Snort [44].

2.4 Artificial Immune Systems

One emerging field in intrusion detection is the use of artificial immune systems (AIS) for detecting attacks. Most of these systems focus on the adaptive portion of the biological immune system. However, in a more general sense, de Castro and Timmis define AIS as “adaptive systems, inspired by theoretical immunology and observed immune functions, principles and models, which are applied to problem solving” [43]. Two main areas in AIS are negative selection based algorithms and algorithms derived from immune network theory. This work focuses on the former area, but incorporates other ideas from immunology.

The goal of AIS used for computer security is to detect attacks by determining “self” from “non-self”. This is accomplished by monitoring certain attributes in the target system. In the negative selection algorithm, a set of normal data contain these attributes is used to train the system about the self data space by emulating biological negative selection.

The attributes used for AIS are usually represented as a string. The types of attributes commonly used are real valued, integer valued, Hamming and symbolic [43]. Hamming and symbolic attributes are sometimes commonly referred to as discrete attributes in certain literature. There can also be a cross-reactivity threshold which allows partial matching between attribute strings. This threshold is expressed as a distance from an attribute value. Any strings with attribute values that fall within this distance are considered a match [43].

The attribute strings for a negative selection based AIS form the non-self space. These strings should not match with self strings, which would be the normal activity for the system. Only attacks which have attribute strings that match one of the AIS’s attribute strings can be detected as attacks. This is roughly analogous to how the biological immune system cells have receptors with certain shapes that can only recognize similarly shaped antigens.

A matching function is defined which allows two attribute strings to be compared. The matching function can return a binary value such as “match” or “not match” or an affinity measure. An affinity measure is essentially the distance between the two strings. The smaller the distance, the strong the affinity is between the two strings.

There are some limitations to the negative selection algorithm. As shown by Gonzales *et al.* [45], the negative selection algorithm is sensitive to the kind of matching function used. They found that several of the binary matching functions presented in works by the University of New Mexico group [20, 16] as well as the Hamming distance matching rule defined by Farmer *et al.* [46] has severe limitations in the non-self coverage space due to the nature of the matching rule. They found that the real valued matching function proposed in [47] achieved far better results than the other matching functions. This led them to conclude that the issue with the non-self coverage space lays with the matching rule and not the negative selection algorithm itself [45].

2.4.1 Affinity Maturation via Genetic Algorithms

The original affinity of an attribute string, or antibody, to an attack string, or antigen, may be low. In biology, the process of affinity maturation results in antibodies with greater affinity to antigens. In an artificial immune system, evolutionary computation is often used for the affinity maturation process. Evolutionary computation a randomized “generate and test” hypothesis space search inspired by the process of evolution in nature. The goal of evolutionary computation is to learn a target concept. Its main components are a fitness function to test hypotheses, a population of individual hypotheses and an abstract genome that represents an hypothesis. The population progresses iteratively through many generations, each new generation derived from the last, before the target concept is learned. Fitness measures how well a hypothesis matches the target concept. The most fit hypotheses are more likely to contribute their solutions to future generations.

In artificial immune systems, the target concept is to have high affinity between antibodies and antigens. The fitness function is therefore the affinity measure. The hypothesis representation is the attribute string. The population is all the attribute strings for the AIS. Typically, one type of evolutionary computation called genetic algorithms is used for affinity maturation. For genetic algorithms, the representations of the hypotheses are called chromosomes and the chromosomes are manipulated for each new generation. The general process is to evaluate fitness, select survivors, select parents, create the new population and mutate the new population. Each individual of the population has its fitness evaluated by the fitness function. A certain percentage of the most fit individuals of the population is probabilistically selected as survivors. Survivors are carried over to the next generation. Pairs of individuals are probabilistically selected based on their fitness to be parents. The parents’ chromosomes are recombined to create two children. The children are added to the next generation.

The crossover operation is the most common method to recombine the parents’ chromosomes. Three types of crossover are single point, two point and uniform. For single point crossover, the parent chromosomes are spliced and swapped at position i to make the two children chromosomes. Two point crossover is similar, but splices the chromosomes

at two points. With uniform crossover, each position in the chromosome randomly and independently decides whether or not to swap with the other chromosome. This results in many crossover points. The end result of the crossover operation is two children which have genetic characteristics of both parents.

The population for the next generation consists of survivors and children. Before the next generation begins, random individuals undergo mutation. The nature of the mutation depends on the structure of the chromosome. For artificial immune systems, the chromosome is a string, so the mutation is a single bit flip at a random point along the string. Mutations are not always beneficial, but when they are, they tend to create new hypotheses that search a nearby area relative to the original hypotheses.

One issue with this population driven evolutionary process is that a very fit individual can disproportionately fill the population with similar hypotheses. This condition is called crowding or over-fitting. The reduced genetic diversity of the population reduces the likelihood of searching new areas in the search space. This can slow or stop progress towards the target concept, possibly leaving the population stranded in a local maxima. Crowding can be mitigated by altering the fitness function to penalize when there are too many similar hypotheses, by restricting which individuals can be chosen as parents or by changing the selection method from probabilistic to rank or tournament selection [48]. For artificial immune systems, crowding can result in an antibody population which only reacts to certain classes of antigens. This can lead to several classes of attacks being undetected by the AIS. In this work, a deterministic rank selection method was used to mitigate the effects of crowding.

2.4.2 Prior Work

Research into artificial immune systems has been occurring for many years. de Castro and Timmis present a good history of artificial immune systems and a survey of AIS related projects [43]. This project was inspired mostly by the works of the Williams *et al.* at the Air Force Institute of Technology [15] and Forrest *et al.* at University of New Mexico [17, 18, 19, 20].

Forrest and others at the University of New Mexico have been working in this field

for many years. Some early work by Forrest [20] in the field focused on distinguishing *self* from *non-self* to detect changes that have occurred on a protected resource. This research laid out the basic concepts of negative selection and partial matching that is used in much of the later research by the University of New Mexico group [20]. Later work draws more heavily on the parallels between biological and artificial immune systems. Basic principles of an immune system and potential areas of application to security are explored in [17]. Forrest and other researchers at University of New Mexico have also developed a network intrusion detection system called LISYS. LISYS monitors the source IP address, destination IP address and port number of TCP SYN network traffic and uses a trace of normally observed traffic as self [18, 19].

The work of Williams *et al.* builds on the LISYS work by expanding the scope of the monitoring. This system, called Computer Defense Immune System (CDIS), monitors up to 28 features of the traffic headers, instead of the 3 features LISYS monitors. LISYS also looks at only the TCP SYN packets while CDIS monitors all TCP, UDP and ICMP packets. TCP, UDP and ICMP are the most common network protocols seen in most networks. CDIS consists of a set of antibodies for each protocol that match on randomly selected features of the protocol. Most of the features of each protocol describe a range of values while the other features are for discrete or boolean values. CDIS was able to achieve low false positive and low false negative rates [15].

While these works do incorporate many features of biological immune systems such as negative selection and affinity maturation, they do not incorporate other features such as collaboration between detectors. The multilevel immune learning algorithm (MILA) presented by Dasgupta *et al.* however does incorporate this collaborative structure. MILA is designed after the interactions of B cells, helper T cells, suppressor T cells and antigen presenting cells in the biological immune system. The experiments presented show MILA to perform better than simple negative selection algorithms which do not incorporate collaboration between detectors [49].

2.5 Approach

This system uses a detection scheme that is somewhat analogous to B cells which do not require helper T cells to be activated. This system focuses solely on the detection of an attack. Even though antibodies in the biological immune system are more associated with directing the response to a pathogen, the term antibody will be used to refer to the detectors since this system does not incorporate the memory cells which are associated with B cells.

The main goal of this system is to detect attacks with a low false positive and a low false negative rate. An ideal system would minimize both of these metrics. This system incorporates the principles of random generation, negative selection, partial matching and affinity maturation.

The antibodies are generated at random initially. Since this random generation can produce an antibody that would react to self, negative selection is used to destroy such antibodies. The self representation may not be complete so some generalization is needed in the reaction of the antibody. This generalization is typically achieved through some sort of partial matching. At the end of this process, an antibody is produced that should only react to non-self. The initial random antibody population may not perform well at detecting non-self, so affinity maturation is used to further hone the performance of the antibodies. A genetic algorithm is used to breed the population over several generations to optimize the detection ability of the population.

One feature also explored that falls a bit outside of the realm of most negative selection based AIS is relaxing some principles of the system to allow fine-grained tuning of the rate of false positives and false negatives. By trading off some of the rigidity of certain processes, a balance can be achieved between these two performance metrics. One tuning parameter is implemented during the negative selection, detailed in Section 2.5.3, which is somewhat analogous to the positive selection of T cells in the biological immune system. The other tuning parameter is implemented during partial matching, described in Section 2.5.2, which implements a rudimentary collaborative system, but without the distinction between B and helper T cells that is present in the biological immune system.

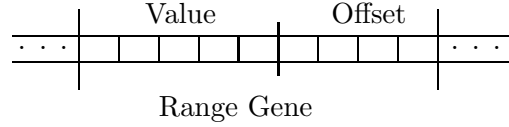


Figure 2.1: An example range valued gene on the chromosome for an antibody, showing the value, offset pair.

These parameters are set by the user before initializing the antibodies.

2.5.1 Antibodies

A fingerprint of the monitored traffic is used as the basis for the antibodies. The fingerprint is a set of attributes associated with the monitored traffic. To support partial matching against test data, some of the attributes are represented as a range value. Range values are similar to an attribute set with a cross-reactivity threshold except that the threshold is specific to each attribute instead of using the same threshold for all attributes. By using a range of values instead of one value, more coverage is achieved, thus supporting partial matching. Each antibody also only checks a subset of the attributes, the expressed attributes. This further supports partial matching since it is rare to observe a request that has all of the attributes.

The remaining attributes are discrete attributes. All discrete attributes are represented as a bitmap. Range attributes are represented as a base value x bits long and offset $x - 1$ bits long. The maximum value of these range attributes was selected based on observations of attack and normal requests. This bit-oriented structure for the antibody facilitates the genetic algorithm operations during the antibody lifecycle. Each attribute is viewed as a gene, as illustrated in Figure 2.1, with only a subset of the genes being expressed. The composition of the attributes in an order specified by the implementation is the chromosome.

2.5.2 Matching Function

When a request is tested to see if it matches an antibody, it is compared against all expressed attributes of the antibody. For a discrete attribute, a match occurs when the

set bit(s) of the request are also set in the antibody. For a range attribute, a request value y matches a range attribute with a base x and an offset o when $x \leq y \leq x + o$. A request is labeled an attack by an antibody when the request matches all expressed attributes of the antibody.

The population of antibodies determine the final label of the request as attack or normal. A request is not considered an attack unless z or more antibodies in the population label it as an attack. This value z is the population agreement threshold. This tuning parameter limits the number of false positives. It is used to offset the effect of the negative selection tuning parameter, detailed in Section 2.5.3.

2.5.3 Lifecycle

The lifecycle is the phases through which each antibody in the current generation of the population go. These phases are: creation, negative selection, testing and breeding. These phases are illustrated in Figure 2.2.

The creation phase is how the antibody enters the current generation. For the initial generation, this is by random creation. During random creation, two to five attributes are randomly selected to be expressed. The values for these expressed attributes are also randomly set. All non-expressed attribute values are set to 0. For succeeding generations, an antibody is created by surviving the prior generation, being the offspring of antibodies in the prior generation or being randomly created to replace an antibody that failed negative selection.

Negative selection uses a subset of the normal data to detect antibodies that react to self. The normal data is probabilistically split into training and testing data to allow for a simulation of “unknown” normal requests during the testing phase. Rather than use the strict definition of negative selection, which would destroy an antibody that matched any request in the training data, a tuning parameter st , the negative selection threshold, is used to determine if an antibody is destroyed. Let n be the number of normal requests that an antibody is tested against. If an antibody labels more than $st * n$ requests, or one request if $st = 0$, as an attack, it is destroyed. When st is greater than zero, antibodies that have a low affinity towards self survive negative selection. This usually causes an increase in

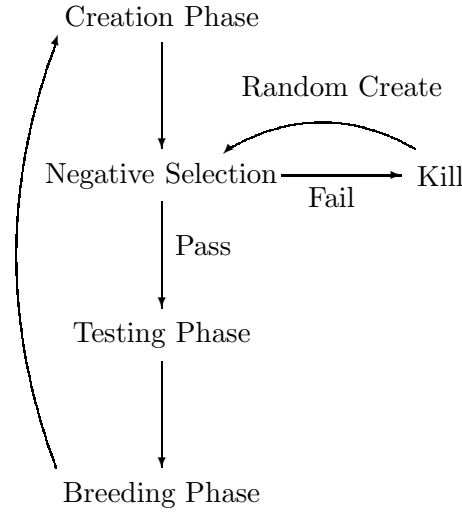


Figure 2.2: Lifecycle of the antibodies

false positives. The hope is that by allowing such antibodies to survive attacks that closely resemble self can be detected. Any destroyed antibody is replaced with a randomly created antibody that also undergoes negative selection.

The testing phase simulates an intrusion detection system being given a sample of traffic to label as attack or normal. The testing phase is used to determine the fitness of the antibodies for the breeding phase. The fitness is defined as the number of attacks the antibodies detect. Thus, the system will select for antibodies that detect the greatest number of attacks. These are typically more generalized antibodies.

The breeding phase uses a genetic algorithm [48] to select survivors and create children to begin the next generation. Details of the implementation of the genetic algorithm are given in Section 2.5.4. The best r percent of the population are mated to create the children. The best $1 - r$ percent of the population are selected as survivors. Once the next generation has been created, members of the next generation have the probability m of having a single bit of an expressed gene flipped. Any antibodies from the current generation that are not part of the survivor set are destroyed at the end of this phase.

Table 2.1: Pseudocode for the selection of the children’s expressed attributes

```

For each gene in chromosome
  If both parents express gene
    Children express gene
  Else
    If either parent expresses
      gene and random check is true
      Children express gene
If children express no genes
  S = union of parents’ expressed genes
  Children express randomly selected
  gene s from S

```

2.5.4 Genetic Algorithm Implementation

The main issue during the mating is how to determine which genes the children will express. If the children expressed only the intersection of genes that both parents expressed, the intersection could be the null set which would cause the children to detect nothing. On the other hand, using the union of expressed genes could quickly lead to children that expressed all genes. As mentioned previously, this would greatly restrict the type of attacks the children could detect since very few attacks would match against antibodies that expressed all genes. To balance these two concerns, the method outlined in Table 2.1 is used.

A single point crossover [48] is used during the mating to create the chromosomes the children inherit from the parents. The crossover point occurs only within an expressed gene of the children. As a consequence of this, if a child does not express a gene, the value of this unexpressed gene is not altered when it is inherited by the child. Since this unexpressed gene may have contributed to the parent’s ability to detect attacks, its preservation in the chromosome of the child allows for the possibility of it being reactivated in later generations during mating with antibodies that do express the gene.

2.6 Implementation for Web Server Attacks

An implementation of this system was tested using web server attacks. This is a desirable form of traffic to test the system on as there is readily available data. A web server

Table 2.2: Number of bits used (offset and value bit for range attributes) and valid values for all antibody attributes

Attribute	Bits	Value or Range
HTTP Command	4	Bit 0 is GET, bit 1 is POST, bit 2 is HEAD, other is bit 3
HTTP Protocol	4	Bit 0 is 0.9, bit 1 is 1.0, bit 2 is 1.1, bit 3 is other
Request Length	17	0 – 766
Number of Variables	5	0 – 10
% Characters	9	0 – 46
' Characters	5	0 – 10
+ Characters	5	0 – 10
.. Characters	7	0 – 22
\ Characters	5	0 – 10
(Characters	5	0 – 10
) Characters	5	0 – 10
// Characters	5	0 – 10
< Characters	5	0 – 10
> Characters	5	0 – 10

request is also relatively easy to abstract into a fingerprint to be used in the system. The attributes used in the fingerprint were selected based on the author's inspection of attack requests and also on the papers announced on a security related Internet forum [50, 51].

The attributes of the fingerprint are summarized in Table 2.2. The HTTP command is a discrete value that can be GET, POST, HEAD or other. GET and POST are the most commonly used HTTP commands. Attackers can attempt to gather information about a website by sending many HEAD commands or uncommon HTTP commands. The HTTP protocol is also a discrete value that can be 0.9, 1.0, 1.1 or other. The current valid HTTP protocols are 0.9, 1.0 and 1.1. An unknown or malformed protocol string could also be an information gathering attack. The length of the request is a rough indicator of an attack attempt. Most requests are fairly short in length and a long length may indicate the presence of malicious code. The number of variables indicates how many parameters were passed by the user to the script. If more variables are seen than typically seen in normal requests, it could be an attack attempt. The % character is used for the various encoding

methods, such as hex encoding for shellcode, and is very common in several classes of attacks. The ' character is used for SQL injection attacks [51]. The + character is interpreted as a space and is often used in directory traversal attacks on Microsoft IIS servers. The .. and \ attributes are often used in directory traversal attacks. The (,), < and > attributes are used in cross site scripting attacks that attempt to inject Javascript into a webpage dynamically created by a script [51]. The // attribute can represent either a subset of a long sequence of / chars which is an attempt to exploit an old Apache vulnerability [51] or an attempt to proxy through the server.

2.6.1 Data Set

Several sources of data were used for this experiment. The logs from a live Internet web server [52] were used to gather normal and attack data. These logs covered the period of Jan. 1, 2002 to Mar. 19, 2002. There were 2166 unique normal request strings and 36 unique attack request strings in these logs. To add additional normal requests, the 1999 Lincoln Laboratory data [53] was used. Only the data from weeks 1 and 3 were used since these weeks contain attack-free data. This yielded 47,885 unique normal requests. Additional attack requests were gathered from proof-of-concept requests posted on the Bugtraq mailing list [54] and from research into web server attacks. 68 unique attack requests were gathered from Bugtraq from Jan. 3, 2002 to Mar. 18, 2002. 80 unique attacks were gathered from the research into web server attacks.

In total, there were 50,040 unique normal request strings and 183 unique attack request strings once duplicates between data sources were eliminated. When running the tests, the normal request data set was probabilistically split, with about 80% being used in the negative selection phase and about 20% used in the testing phase. Due to the probabilistic nature of the split, it was possible that a normal request string would be in both the negative selection data set and the testing data set. All of the attack request strings were used in the testing phase.

2.6.2 Testing Parameters

The parameters for testing were as follows. Population sizes (p) were 125, 250, 500 and 1000. Mating percentages (r) were 0.3, 0.5 and 0.7. Mutation rates (m) were 0.05 and 0.1. Negative selection thresholds (st) were 0.0, 0.0001 and 0.0002. Population agreement thresholds (z) were 1, 2 and 3.

The case where $st = 0$ is considered to be the baseline, as this causes the system to run in the standard negative selection mode where any antibody that reacts to any instance in the self data set is destroyed. When $st = 0$ and $z = 1$, the false positive rate is nearly zero. Since the purpose of raising z is to lower the false positive rate, no tests were conducted for $st = 0$ and $z = 2$ or $z = 3$. For all other values of st , tests were run on all possible combinations of st and z .

For all tests, ten populations of antibodies were tested in parallel. The best performing generation of antibodies was retained by the system to counteract the effects of overfitting, which tends to begin occurring after about a half dozen generations. The results from the best performing generation for each population are used in the analysis. Results for false positive and false negatives are expressed as the actual number of false positives and false negatives in the graphs. The results for each test were averaged and 95% confidence intervals were computed using the t distribution.

2.6.3 Results

The following variables had no significant effect on the results. Although the mutation rate $m = 0.05$ did on average yield lower false positive and false negative rates than $m = 0.1$, this difference was not significant. Similarly, a higher mating percentage did on average yield lower false negatives and higher false positives, but this was only significant in a fraction of the tests.

Several significant trends emerged that held across all tests. Larger populations on average had a lower false negative rate, but also a higher false positive rate. This difference was significant when comparing $p = 1000$ to $p = 125$, except with the baseline tests. The baseline tests did not have a significant difference between the false positive rates since the

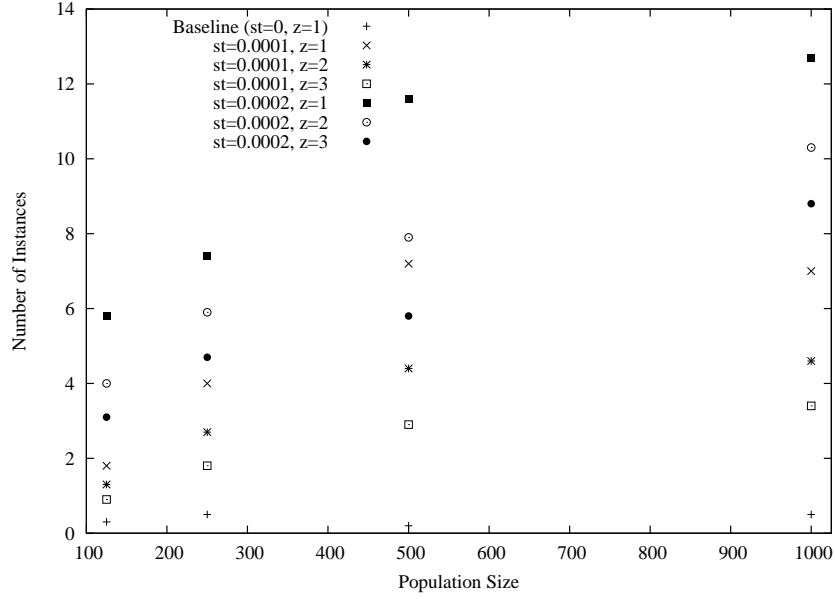


Figure 2.3: False positive trends across all population sizes for $r=0.7$ and $m=0.1$

false positive rate was nearly zero in all cases. The difference was not significant for all tests when comparing other population sizes. These trends are illustrated in Figure 2.3 for false positives and Figure 2.4 for false negatives.

The negative selection threshold also had a significant effect on the number of false positives and false negatives. As expected, when st increased, it was more likely that an antibody that reacted slightly to the self data set would pass the negative selection phase. This led to having more false positives since these antibodies were more likely to react to self in the testing phase. However, it also led to fewer false negatives since some attacks had similar fingerprints to some normal requests. In the most rigid case of $st = 0$, antibodies that could detect such attacks would be destroyed. As st rose, these antibodies would survive and detect the attack. With the exception of a few tests, this difference was significant. However, when the difference was not significant, higher st values always had lower average false negatives and higher average false positives.

The population agreement threshold meant that z antibodies in the population had to label the request an attack before it would be considered an attack. This threshold works as a countermeasure to the higher false positive rate that a higher negative selection threshold causes. As z increased, more antibodies had to agree that a request was an attack

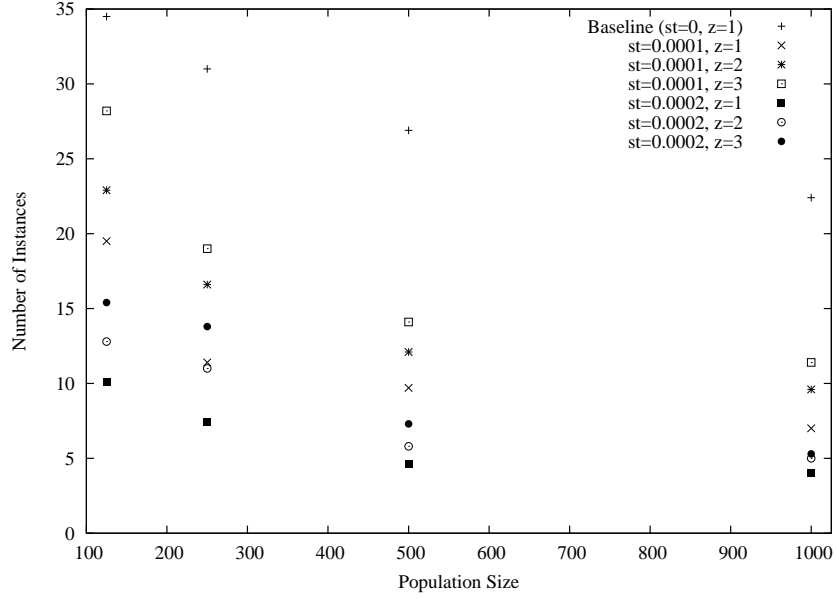


Figure 2.4: False negative trends across all population sizes for $r=0.7$ and $m=0.1$

before it was labeled an attack. So as z increased, the average false negative rate was higher and the average false positive rate was lower. This difference was significant in most cases when comparing $z = 1$ to $z = 3$, but $z = 2$ was not significantly different than either $z = 1$ or $z = 3$ in most cases.

An interesting note about the results is that the population agreement threshold does not completely counteract the negative selection threshold. This is a desired effect. In the baseline case, where $st = 0$ and $z = 1$, the false positive rate is rather high, ranging from 12% to 29%. Even though there is an almost zero false negative rate, the number of attacks missed may be considered unacceptable for certain web servers. By adjusting these two thresholds, an operator can strike a balance between false positives and false negatives. The best tests achieved a false negative rate of less than 5% and a false positive rate that was less than 6% of all alerts and less than 0.02% of all normal requests. Figure 2.5 illustrates the effects caused by varying st and z when the population size is 1000, $r = 0.7$ and $m = 0.1$.


```
u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b0
0%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0
```

Both requests are processed into the following fingerprint:

```
GET 1.0 375 1 23 0 0 0 0 0 0 0 0 0
```

Most of the antibodies in this test were able to detect the attack after several generations of affinity maturation. Only the smaller population sizes, primarily $p = 125$, were missing this attack in the final generation. For $p = 500$ and $p = 1000$, the attack was always detected in the final generation.

2.7 Conclusion

The aim of this work was to create an artificial immune system for web server attacks that can achieve both a low false positive and a low false negative rate. Using a genetic algorithm to breed these antibodies results in an excellent false negative rate, but a somewhat high false positive rate in the standard artificial immune system model. By using the negative selection and population agreement thresholds, a balance can be struck between the acceptable false positive and false negative rates. This gives the operator the freedom to decide whether or not a higher false positive rate is acceptable in order to detect more attacks. Through this system, an administrator can dynamically adjust a setting to provide more sensitivity to possible attacks at the expense of a higher false positive rate.

Chapter 3

Attack Graphs: Overview of Concepts

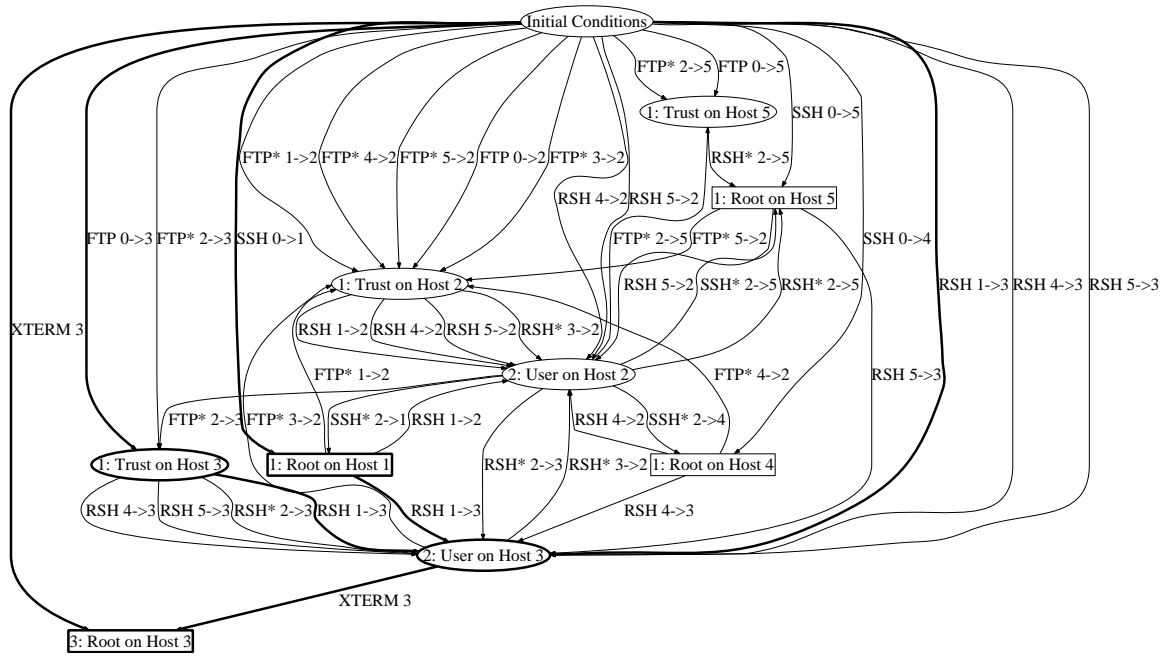
Most vulnerability scanners only evaluate individual machines without considering the network as a whole. Attack graphs are a tool to evaluate the composition of vulnerabilities across the entire network. An attacker could use such compositions to achieve further penetration into the network than he might if he were only compromising individual machines. For example, an attacker could penetrate a public server, and use that as a platform to attack internal servers he would otherwise not be able to reach. This is a classic “foothold” situation where the network has publicly available machines, internal machines that are not directly accessible by the public and a communication channel between the public and internal machines. The attacker compromises a public machine to gain access to the communication channel between the public and the internal machines. He then uses that communication channel to compromise the internal machines. Attack graphs can show such “foothold” situations and thus are useful at several stages of network planning and management. Attack graphs can be used to evaluate new network designs to see which provides more security, assist patch management by determining the critical set of vulnerabilities with respects to mission resources, identify what steps an attacker most likely took during a forensics evaluation, correlate intrusion detection system (IDS) alerts into an overall attack pattern and determine automated responses to a detected, ongoing

attack.

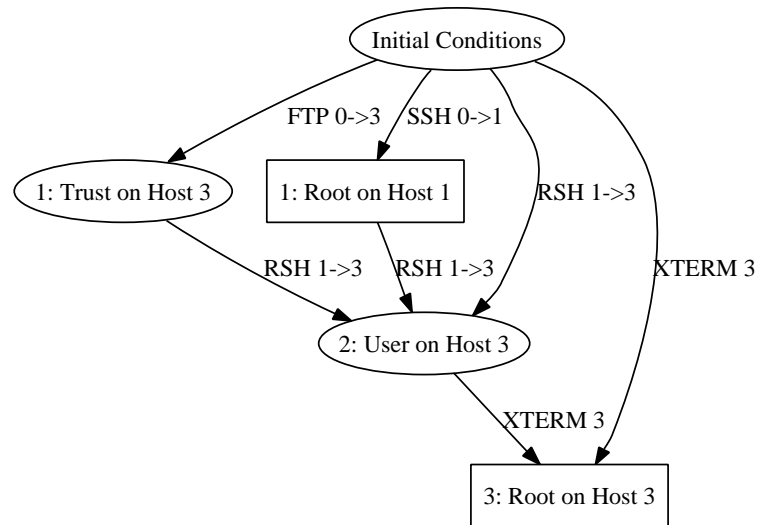
Attack graphs represent exploit paths an attacker can take through the network. Originally the graphs were created by hand during network analysis by a team of experts. These hand-created graphs were often quite large and prone to human error. Several methods [1, 3, 2, 22, 23, 24, 25, 26] have been proposed to generate attack graphs automatically. Automatic methods typically have a library of potential exploits, and they match these attack templates to the properties of the network being evaluated. Typically the templates are represented in a “requires/provides” [21] which defines a set of preconditions required for the exploit to be executed and a set of postconditions defining the consequences of the exploit being executed. The automatic methods repeatedly find matches for preconditions of the templates and apply the postconditions until a stopping condition is met or no more matches are possible. This allows the discovery of scenario attacks, where a sequence of exploits are used to achieve an attacker’s goal, and of “foothold” situations.

Previous methods either did not scale well to large networks such as with model checking approaches or they required extensive hand coding using graph-based algorithms. This approach uses the expert system Java Expert System Shell (JESS) [5] to generate the attack graph. JESS is appealing for attack graph generation because it has an efficient pattern matching search algorithm. JESS also has a scripting rule language that allows for new attack templates to be added without requiring extensive recoding of the entire system. Compared to model checking based approaches, this approach is able to evaluate much larger networks with less computational and memory requirements due to not enumerating all possible combinations of variables. The analyst may change the method of visualizing the graphs based on his needs. The tactical view allows the analyst to see all possible paths while the strategic view lets the analyst see if there exists a way for the attacker to compromise resources. This approach also supports further abstraction that will further increase scalability, such as abstract exploit classes and clustering of similar hosts that are detailed in Chapter 5.

Figure 3.1: Tactical and strategic attack graph views for a five server network described. The highlighted path represents one path to get root on Host 3. The highlighted path is shown in Figure 3.2(b).



(a) Hybrid



(b) Highlighted Path

Figure 3.2: Hybrid attack graph view and the highlighted path for a five server network. The hybrid view shows all edges involving Host 2 as either a source or target machine.

3.1 Attack Graphs

Attack graphs are used to represent the multiple exploit paths attackers can take through the network. As an attacker executes exploits, he gains capabilities such as root privilege level on a host. These additional capabilities may allow an attacker to execute an exploit that he could not have executed initially. Vulnerability scanners are unable to model this facet of attacker behavior, but attack graphs can. Important challenges for attack graphs that are addressed in this work are computing the exploit paths, visualizing the graph and analyzing the graph based on intended use of the attack graph.

3.1.1 Computation

The automatic computation of the exploit paths is typically accomplished by matching attack templates to the properties in the network being evaluated, such as vulnerabilities on a host. An attack template is a potential exploit of a vulnerability, also called an *atomic attack*. An example of an attack template would be an SSH daemon buffer overflow that allows a remote attacker to obtain root privileges on the server.

Atomic attacks are defined in a requires/provides format [21]. In this format, there is a set of preconditions and a set of postconditions associated with each attack template. The preconditions must be true before the atomic attack can be executed. The preconditions can represent concepts such as what vulnerability is needed for the attack to succeed or the attacker privileges on the target and/or source machine. The postconditions represent the new state of the network that has resulted due to the execution of the atomic attack. The postconditions can provide new capabilities that can enable other atomic attacks that the attacker could not previously attempt. Using the previous SSH daemon example, the preconditions would be that the target machine is running a vulnerable version of the SSH daemon, the source machine can connect to the target machine and the attacker has user privileges on the source machine. The postconditions would be that the attacker now has root privileges on the target machine.

In a *monotonic* implementation of attack graph generation, the postconditions can only add capabilities to the system or do nothing. In a *non-monotonic* implementa-

tion, the postconditions can also remove capabilities, such as stopping a particular network service. Denial of service (DoS) is a classic example of a non-monotonic attack as a previously available service has now been rendered unavailable. However, such an attack can be converted to a monotonic format by viewing the “DoS” state as an additional capability in the system that prevents access to a specific resource rather than removing the resource as one would do in a non-monotonic implementation. Monotonicity also requires that the preconditions be conjoined, with exploits that have disjoint preconditions modeled in separate atomic attacks. Several previous works argue that monotonicity is the key to ensure scalability of attack graph generation by reducing the complexity of the model from exponential to quadratic [23, 27]. This method is currently monotonic, although the JESS language would support a non-monotonic implementation. Non-monotonicity is limited to maintain scalability.

A sequence of atomic attacks comprises an exploit path. The attack graph consists of one or more exploit paths. Traditionally, all exploit paths terminate at a specific goal or set of goals, such as obtaining root on a certain host. This *goal-driven* approach allows the analysts to specify the negative effects they do not want to occur on the network. However, for certain uses such as evaluating attacker penetration into the network, an *exploratory* graph that continues until the attacker runs out of possible atomic attacks is more desirable. Some automated methods [1, 3, 2, 22] use backwards chaining, where the search is performed backwards from the goal(s), or counter example generation, where only paths that lead to the goal(s) are returned, so they are only able to generate goal-driven attack graphs. Other methods [23, 25, 28], including this method, use forward searches from the initial state. Forward searching methods are capable of generating both exploratory and goal-driven graphs although previous work has focused on goal-driven graphs by pruning the exploit paths that do not lead to the goal.

In order to automatically generate attack graphs, one needs a description of the network, the initial attacker capabilities and an attack template library. In previous work [1, 3, 2] and in this model, the network description contains the topology of the network, firewall rules, intrusion detection system rules, a list of hosts, the services offered on the hosts and the vulnerabilities detected on the hosts. Other properties of the network, such

as responses, could also be added to extend the analysis. The attacker capabilities describe the initial capabilities the attacker has on the network such as his privileges on each host. Other previous methods [24, 25, 22, 23, 26] use similar data for the network description and attacker capabilities, but differ in the level of detail. With this information about the initial conditions of the network and the attack template library, the matching algorithm can compute exploit paths through the network using either goal-driven or exploratory methods.

One of the limiting factors in automatic attack graph generation is obtaining the network description and writing the attack template library. Automated network discovery tools and vulnerability scanners do not give a complete view of the network structure and machine vulnerabilities. Firewalls may prevent these tools from seeing certain aspects of the network structure or certain vulnerabilities on machines. The availability of personal firewalls also makes it difficult to determine the firewall rules present in the network even if all the rules for the core routers are known. Personal firewalls on user-controlled machines may also prevent an administrator from scanning the machine with automated tools. The ability of attack graphs to perform “what if” scenarios, where the analyst or administrator can guess certain unknown aspects of the network, helps minimize the effects of the unavailability of complete data by automated methods. The data gathered by such tools can also be embellished by an expert before being analyzed by the attack graph tool. An expert is also currently required to convert known exploits into the attack template library notation.

3.1.2 Visualization

In this method, the visualization of the attack graph is a separate component from the generation of the exploit paths. Currently, JESS computes the exploit paths and *dot*, from the Graphviz project [55], is used to visualize the graph. Representing the exploit paths is the main issue in visualization. Previous works have used several different methods. In [1, 3, 2], nodes represent the preconditions that are satisfied for each exploit and the edge leaving the node represents the execution of the exploit. Each possible combination of capabilities is represented by a separate node, leading to an explosion in the number of nodes for larger networks. Noel and Jajodia [27] call this type of visualization a *state*

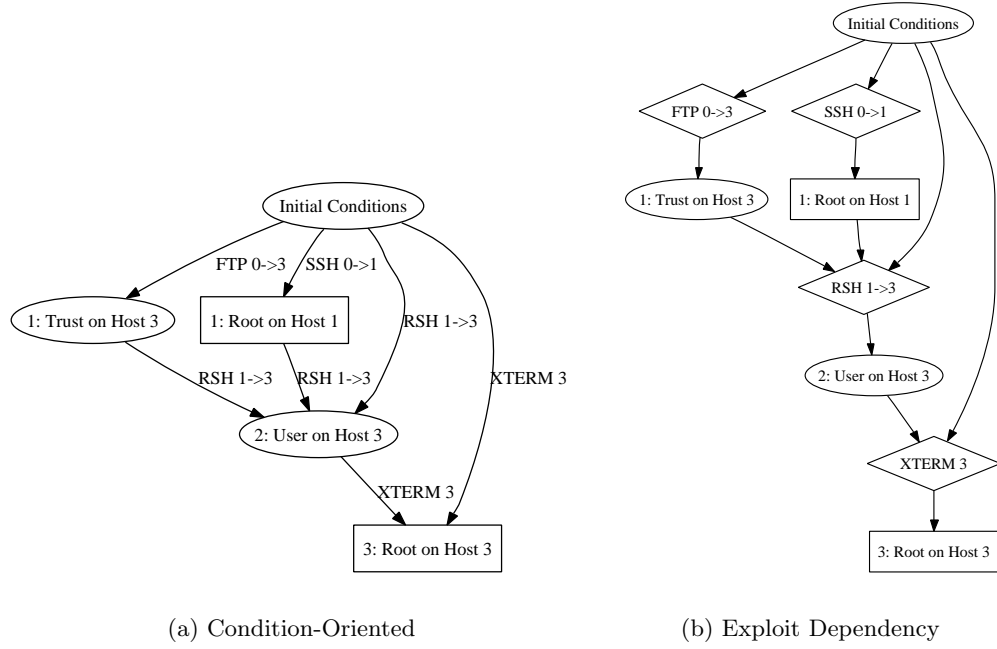


Figure 3.3: Condition-oriented and exploit dependency based graphs for the highlighted path show in Figure 3.2(b).

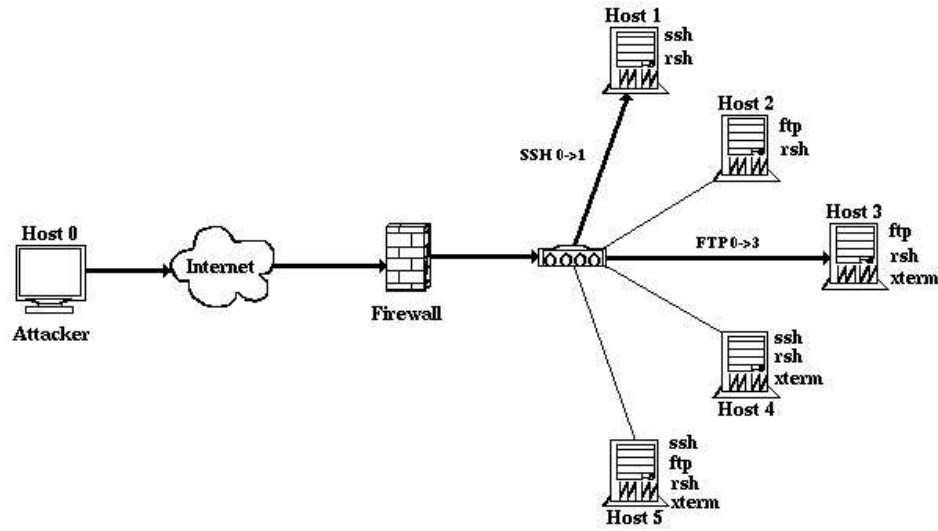
enumeration graph. In [23], nodes represent individual capabilities and the edges correspond to the execution of exploits. There is one node per capability and potentially multiple edges per execution of an attack template. Noel and Jajodia [27] call this type of visualization a *condition-oriented dependency graph*. In [28, 26, 27], nodes represent both capabilities and exploit executions, while edges are merely showing dependencies between nodes. Thus there is one node per capability and one node per execution of an exploit. They call this type of graph an *exploit dependency graph*. By default, the graphs are condition-oriented dependency graphs, but they can also be converted to exploit dependency graphs. Figure 3.3 shows both condition-oriented and exploit dependency views for the highlighted path from Figure 3.2(b).

This method further supports filtering the exploit paths within the graph to assist the analyst in visualizing the attack graph. Specific edges on the attack graph can be hidden in order to reduce the complexity of the visualization of the graph. This can be done automatically via two main viewing modes, “tactical” and “strategic” views, or interactively by the user. The tactical view shows all possible exploit paths through the network. This is

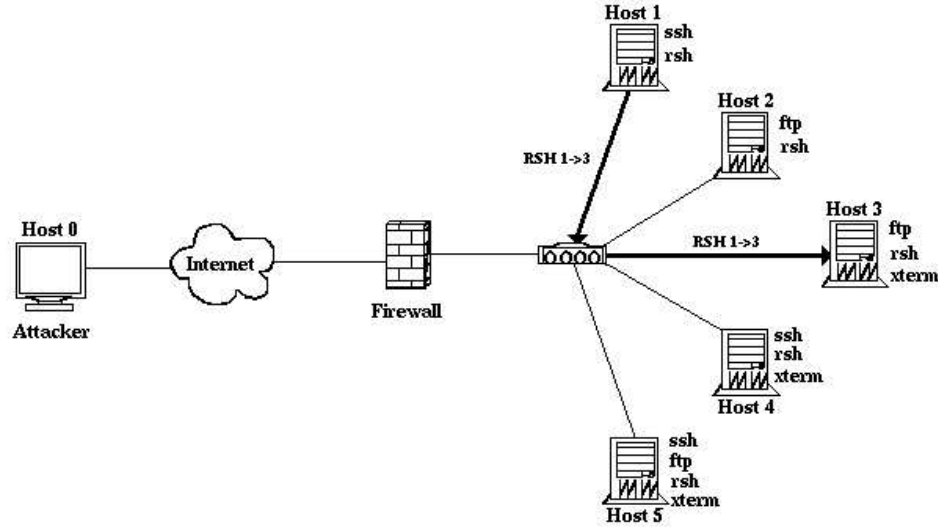
useful for situations which require knowledge of all possible ways an attacker might obtain a specific capability. The strategic view is a “there exists” view which shows all capabilities an attacker can obtain, but not all possible ways in which he can obtain each capability. Thus it shows only that there exists a way for the attacker to obtain a capability. Both views for any given network will have the same capability nodes. Through a user-defined filter a hybrid view between tactical and strategic views can be generated. An analyst could show all possible exploit edges for certain capability nodes (tactical view) while showing only specific exploit edges for other nodes (strategic view).

Figures 3.1(a) through 3.2(a) show an example of the filtering views of an attack graph for a five server “DMZ” scenario. The network consists of five publicly accessible machines in a demilitarized zone (DMZ). The network topology and the vulnerabilities present on each machine is shown in Figure 3.4. This network was further expanded by adding internal machines for the scalability tests presented in Section 4.6.1. Elliptical nodes are used for capabilities, with rectangular nodes used to denote the root privilege capability. All capabilities in the network description and attacker profile are represented by one “Initial” node for simplicity, although the individual capabilities could have also been graphed. The exploit path shown in Figure 3.2(b) has been highlighted in all three views. This highlighted path is one path by which the attacker can get root privileges on Host 3. In the first level, shown in Figure 3.4(a), the attacker executes two attacks: FTP from his machine against Host 3 and SSH from his machine against Host 1. The FTP attack establishes a trust relationship on Host 3. The SSH attack gives the attacker root privileges on Host 1. In the second level, shown in Figure 3.4(b), the attacker uses the two capabilities he gained on the first level to launch the RSH attack from Host 1 against Host 3. This gives him user privileges on Host 3. In the third level, shown in Figure 3.5(a), he uses the XTERM local buffer overflow on Host 3 to escalate his privileges on Host 3 from user to root. The tactical view, Figure 3.1(a), shows all the exploit paths in the network. The strategic view, Figure 3.1(b), shows a subset of the edges from the tactical view. This subset corresponds to the execution of the “main” rules as defined in Section 4.4. The hybrid view, Figure 3.2(a), shows all exploit edges involving Host 2.

The advantage of using the filtering views is that the visualization can be cus-

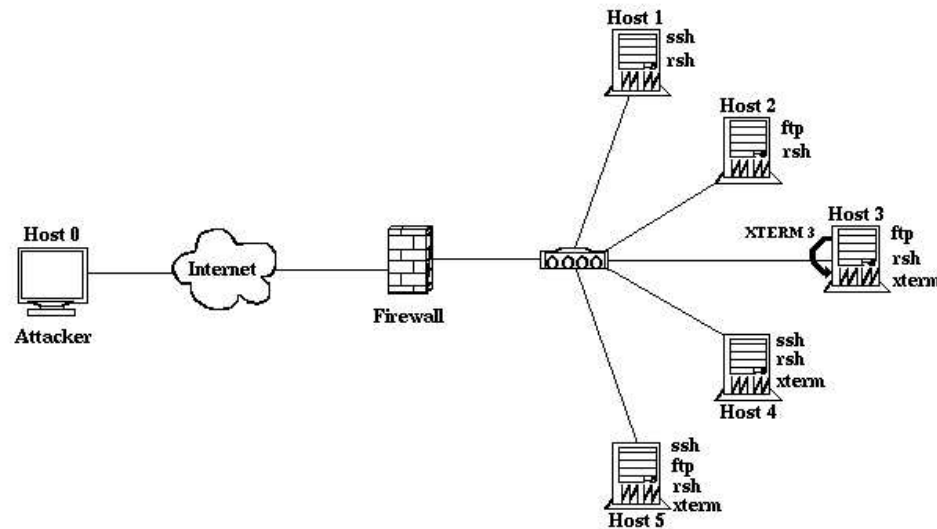


(a) Level 1



(b) Level 2

Figure 3.4: Level by level breakdown of the attack sequence in the highlighted path of Figure 3.2(b). In level 1, two attacks, SSH from 0 to 1 and FTP from 0 to 3, are executed. In level 2, one attack, RSH from 1 to 3, is executed.



(a) Level 3

Figure 3.5: Level by level breakdown of the attack sequence in the highlighted path of Figure 3.2(b). In level 3, one attack, XTERM overflow on 3, is executed, giving the attacker root on Host 3.

tomized for the needs of the analyst. If the analysis does not require seeing all the edges, they can be hidden. This reduces the visual complexity of the graph.

3.1.3 Uses of Attack Graphs

Attack graphs are useful in the evaluation of the security of a network by providing a composition of atomic attack steps. This might be overlooked in a review of vulnerabilities present on individual machines because the interaction of several vulnerabilities might not be considered. For example, a server running a vulnerable version of the SSH daemon might be thought to be harmless because it is behind a firewall so external attackers could not connect to it. However, if there is also a vulnerable public FTP server that is allowed to connect to the SSH server, an attacker could compromise the FTP server and use the FTP server to compromise the SSH server, bypassing the firewall that would prevent him from attacking the SSH server directly. Attack graphs allow administrators to see such interactions. There are several situations in which attack graphs are helpful: patch management,

network design, forensics analysis, mission analysis, IDS alert correlation and real-time IDS monitoring.

The goal of patch management is to determine the most desirable set of patches required to secure the network. In this situation, a tactical view would be most appropriate as the analyst is interested in all possible exploit paths. For patch management, the graph analysis would focus on finding the minimum cut of edges that would limit or prevent an attacker from penetrating a network. This set of edges would represent a set of atomic attacks on specific hosts. By evaluating the preconditions of those atomic attacks, the analyst can derive a set of vulnerabilities that require patching. Several previous works have done this sort of analysis [24, 3, 1, 2, 23, 26, 28]. For network design, the analyst is more interested in “what if” scenarios and limiting attacker penetration. This might involve generating a graph of theoretical vulnerabilities or comparing various firewall rulesets. In this situation, a strategic view would be more useful. By comparing multiple strategic views corresponding to the different designs being considered, the analyst can determine which design most limits the capabilities the attacker can theoretically obtain.

With forensics evaluation, attack graphs can be compared to gathered forensics data to determine the exploit path(s) the attacker likely took to compromise the system and what other resources the attacker may have compromised that have not yet been discovered. The tactical view would be most useful for these purposes. For mission analysis, the goal is to determine if mission-critical resources are adequately protected with the current state of the network. The strategic view would be sufficient to show if the mission-critical resources can be compromised. If the strategic view shows that mission-critical resources can be compromised, the analyst may wish to switch to a hybrid view which shows all possible ways the selected resource(s) can be compromised. This allows the analyst to determine a course of action.

For both IDS alert correlation and real-time IDS monitoring, the attack graphs are used to compose multiple alerts into a likely attack scenario. Alert correlation is typically done from logs after an event has occurred and is used to reduce the individual alerts into an overall attack scenario report. A hybrid view which shows all the edges related to the detected alerts would be suited for alert correlation. The attack graphs can also

help determine what other resources the attacker might have compromised that the IDS did not detect. The steps can be listed in the report so an analyst can further investigate if the attacker took those steps. With real-time monitoring, the analyst can be alerted to the next likely steps the attacker will take so that those resources can be protected or monitored more closely. The tactical view would be more appropriate as it would show all possible next steps the attacker could take.

The needs of the analyst can be best met by using various methods of visualization, such as the filtered views presented in Section 3.1.2. Strategic views are well suited for when the analyst needs to determine the scope of the penetration without all the details. The tactical views allow the analyst to see all possible ways in which the attacker can move through the network and is suited for when defensive measures need to be selected.

3.2 Related Work

Several groups have developed tools to automate the generation of attack graphs. These approaches have generally used either graph theory or model checking as the basis of their methods. For the model checking approach, there is an explicit attacker goal, such as compromising a specific machine in the network. Graph based methods do not require a goal unless backwards chaining is used. Goal nodes can be declared in graph-based methods to limit the search.

Swiler, *et al.* [24, 25] used a hand coded tool that takes a library of attack templates and compares the templates to the network and attacker descriptions to create the attack graph. Graph theory methods were used to evaluate the resulting graph for information such as the near-optimal set of paths. They also proposed several graph theory methods in [25] to improve the scalability of their tool, but the methods were not fully implemented.

Ritchey and Ammann [22] used the model checker SMV [56] as the basis of their tool. They hand coded the model in the SMV language but they indicated that automated methods could be developed to translate network and attacker descriptions into the SMV language. Explosion of state space as the network size grew was a major issue with the scalability of their approach.

Jha, *et al.* [3] and Sheyner, *et al.* [1, 2] also used model checking as the basis of their model. They used a modified form of NuSMV [57] that would generate all possible counter-examples instead of just one counter-example. The counter-examples were used to build the attack graph. They also presented two post-processing methods. Minimization analysis finds the critical set of atomic attacks needed for the attacker to achieve his goal. Probabilistic analysis computes the reliability of the network, which is the likelihood that the attacker will fail to achieve his goal. Their methods did not scale well due to the state explosion inherent to model checking methods.

Ammann *et al.* [23] attempted to address the scalability issues with model checking methods by providing a graph-based method. Their method uses a level based breadth-first approach to compute the attack steps. The scalability of their approach is based on the assumption of monotonicity, that is that the attacker is always gaining additional capabilities and never loses capabilities he had previously. With this assumption, the complexity of their model becomes polynomial, as opposed to exponential with model checking approaches. The assumption of monotonicity cannot be assured for all possible exploits, but they argue that many exploits can be abstracted into a monotonic form. Their method also allows the results to be analyzed without explicitly building an attack graph.

In Jajodia, Noel and O’Berry [26], a monotonic, graph-based method similar to Ammann *et al.* [23] is used. It is goal-driven and uses exploit dependency graphs for visualization and analysis. A hierarchical framework is used for the network description to support a wide variety of attack models. The primary use of their method is to determine which initial properties of the network can be manipulated to prevent the attacker from achieving his goal, also known as network hardening. Their analysis consists of representing exploit paths as algebraic expressions involving the initial network properties and determining which properties could be altered to harden the network. In Noel *et al.* [28], the analysis further converts the algebraic expression into conjunctive normal form to derive all possible ways the initial properties can be altered. They then describe a method to select the hardening measure with minimal impact and cost. In [27], Noel and Jajodia also present a method to automatically aggregate nodes using a hierarchy of rules when visualizing exploit dependency graphs.

3.3 Conclusion

Attack graphs are a collection of exploit paths an attack can take within the network. As an attack compromises machines, he gains resources which may be used to compromise additional machines. Vulnerability scanners cannot model this facet of attacker behavior, but attack graphs can. The computation of an attack graph determines these exploit paths by matching attack templates to properties in the network. The computation can be goal-driven, where there is a specific attacker goal that the analyst wishes to prevent, or exploratory, where there is no specific attacker goal. The visualization of the attack graph allows the analyst to see the exploit paths that have been computed.

Attack graphs have several uses for network administrators. A common use for attack graphs is to help determine a set of patches to apply to the system. Vulnerability scanners only tell an administrator that a vulnerability is present in the system while attack graphs tell the administrator the severity of that vulnerability by showing what exploit paths it enables. Attack graphs can also be used as a part of the network design process. In this capacity, the administrator wishes to design her network to limit attacker penetration. Attack graphs can be used to evaluate several potential network designs to see which limits an attacker the most. Attack graphs can also be used as a part of forensics evaluation. The forensics data can be compared to the attack graph to see what paths the attacker likely took through the network. This shows not only how an attacker may have compromised a machine but also what other machines the attacker could have compromised from that point. Similarly, attack graphs can be used for intrusion response by determining what further machines the attacker could compromise, what methods he could use for those future compromises and how to prevent him from taking those exploit paths.

Chapter 4

Attack Graphs: Formalization and Mechanization

4.1 Introduction

The generation of attack graphs, as overviewed in Section 3.1.1, matches attack templates that are in a “requires/provides” format to the conditions that exist in the network. This type of search is well-suited to be performed by an expert system. An expert system takes a set of facts and matches them to the preconditions of a set of rules. The Java Expert System Shell (JESS) was chosen as the expert system to be used due to the efficiency of the Rete algorithm that JESS uses. The Rete algorithm is most useful when the ruleset is static and a small percentage of facts are altered with each rule. This matches well with the nature of attack graph computation. The execution of a single exploit (an expert system rule) alters only a handful of capabilities (expert system facts).

This chapter details the method by which an expert system was used to generate attack graphs. For this chapter, each exploit was explicitly modeled. Chapter 5 describes another model where the exploits are abstracted into abstract exploit classes. In Section 4.2, justifications for using JESS and the Rete algorithm for the computation of attack graphs are given. Section 4.3 details the formal model of an attack graph. The methodology used to compute attack graphs with JESS is given in Section 4.4. Section 4.5 gives theoretical

complexity of computing an attack graph with the Rete algorithm. Section 4.6 has the exploit-based model implementation and experimental results with this model. Section 4.7 gives the conclusions.

4.2 JESS Overview

Java Expert System Shell, JESS, is an expert system that uses the Rete algorithm [4] to match facts to rules. For attack graphs, the rules correspond to the atomic attacks and facts correspond to capabilities and initial properties of the network. The left hand side, LHS, of a JESS rule is the pattern to search for in the fact base and the right hand side, RHS, is the action to take when the pattern is found. For use with attack graph generation, the LHS expresses the atomic attack preconditions and the RHS creates the postconditions. The Rete algorithm is useful when rulesets are mostly static and only a small percentage of facts are added or deleted when a rule is executed.

A brute force search method might iterate through all the facts and rules to find matches on the LHS of the rules each time it executes a rule. This approach is not scalable as the size of the fact base and ruleset increases. It is also redundant in situations where the execution of a rule does not greatly change the fact base. For example, consider a scenario where on pass 1 the system evaluates the LHS of an unexecuted rule to be unmet by the fact base and the executed rule does not add facts that affect this LHS. The brute force method would still evaluate this LHS in pass 2 even though there has been no change which would affect it. This lack of memory between passes makes the brute force method highly inefficient [5].

A Rete network compactly represents the LHS of the rules and removes much of the inefficiency present in brute force search. Nodes in a Rete network typically have either one or two inputs from other nodes and one output to another node. The basic node types are input, join and output nodes. Input nodes tokenize facts and perform basic pattern matching on the facts. Output nodes link to the RHS of a rule and are only reached if all the conditions in the LHS have been satisfied. Join nodes represent the logical structure of the LHS of the rules. A join node receives two inputs and its purpose is to match these

inputs according to the LHS of the rule [4, 5].

A naive method to build the Rete network might be to build the nodes for each rule individually. This can lead to redundant nodes in cases where rules share the same input or use the same logical check. JESS uses node sharing to merge such redundant nodes into one node [5].

Much of the efficiency of a Rete network comes from the memory of partial matches that is maintained within each join node. For example, when a join node receives a new token on its left input, it checks its right memory for a match and stores the token in its left memory. If a match is found, it sends a new token representing this match through its output. With well-written rules, the partial match memory is much smaller than the total fact base size and contains only facts relevant to the node's pattern. So searching for matches within the partial memory is on average much more efficient than searching all the facts as is done with brute force searching.

The Rete network only updates partial match memories when there are new, deleted or modified facts. Also, due to the trickle down design, these facts are only evaluated against rules which might use them, rather than all the rules as in the brute force method. This means the Rete algorithm performs best when there is not a high rate of change in the fact base as this would cause constant updates to the partial match memories of each node.

This makes the Rete algorithm appealing for the attack graph problem. In general, the fact base does not change much when a single atomic attack is executed. For example, with a small three node network such as presented in [1, 3, 2] the execution of one of the atomic attacks modifies, deletes or changes approximately 5-10% of the total fact base. As the size of the network increases, this rate of change per atomic attack becomes even smaller because the initial fact base is larger. This small rate of change makes the attack graph problem well suited for the Rete algorithm.

There are some issues which affect the efficiency of JESS. Two important ones are LHS variable tests and LHS statement order. JESS provides two methods to test the value of a variable in the LHS statement: the *test* command and an inline comparison. Both methods are logically equivalent, but they result in different Rete networks. The *test*

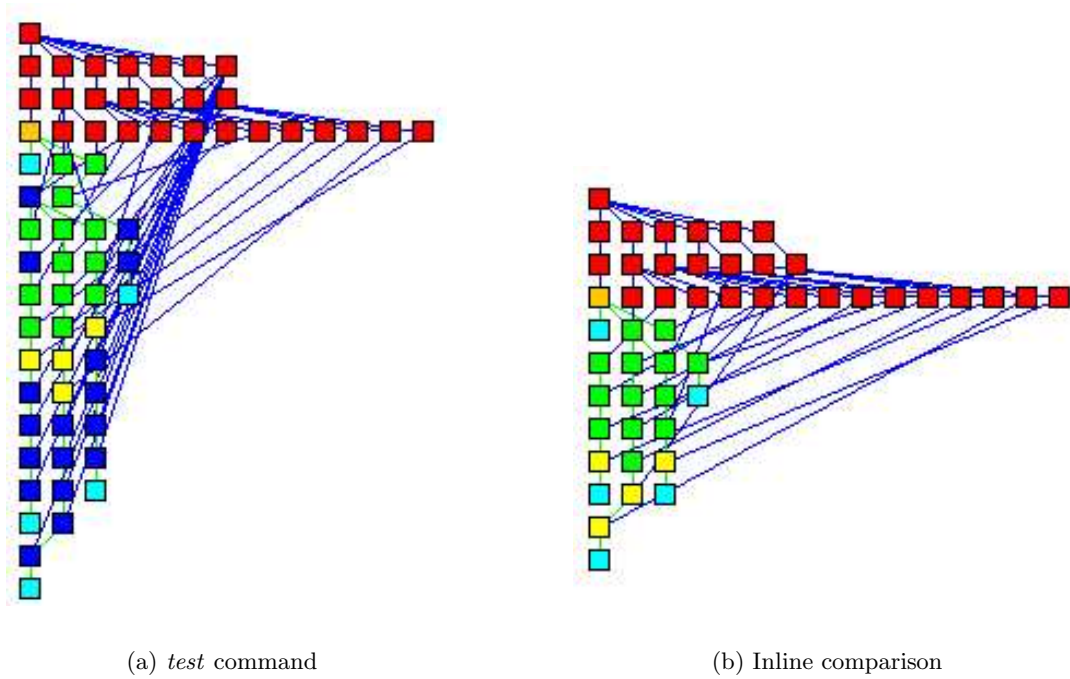


Figure 4.1: Comparing the effects of the different LHS value test methods on the resulting Rete network. The inline comparison results in a more compact and efficient Rete network.

command is implemented by adding join nodes to the network, while the inline method is implemented internally to the input nodes. The difference in the resulting networks can be seen by comparing the graphs in Figure 4.1. Figure 4.1(a) shows the network that results from using the *test* command. Figure 4.1(b) shows the same code, but using the inline comparison. The inline comparison results in a more compact Rete network which is more efficient.

The order in which the preconditions are checked on the LHS can also affect the optimization of using a Rete network. The preconditions which would most likely reduce the number of matching facts selected from the fact base should come first. This will reduce the number of tokens passed to nodes farther down the network and thus limit the size of the partial match memories. This is similar to how one optimizes *if* statements to take advantage of short circuit logic, where the checks most likely to stop the evaluation are put first. If the most restricting preconditions are put higher in the network, the reduction in partial match memories further down in the network will lead to faster evaluation and less memory usage.

Rete is well suited for the attack graph problem since it provides an efficient method of finding matches compared to a brute force method. One must make effective coding choices to gain the full efficiency of Rete as shown by the inline versus test issue and the effects of evaluation order.

4.3 Formal Model

Let N be the set of machines in the network being modeled and C be the set of capabilities an attacker may acquire on each machine. An attack graph is $AG = (V, F, A, V^0, V^F, G, L)$ where:

- V is a matrix of capabilities in the network, $N \times C \rightarrow Boolean$
- F is the firewall matrix, $N \times N \rightarrow Boolean$
- A is the set of atomic attacks, $V \rightarrow L$
- V^0 is the initial capabilities in the network, $V^0 \subseteq V$
- V^F is the final capabilities in the network, $V^F \subseteq V$
- G is a boolean expression concerning V
- L is a list of edge labels, $L \subseteq A \times N \times N$

The capabilities matrix, V , defines the capabilities of each machine in the network. Capabilities are either present or absent in this model, so they are represented by a Boolean. The firewall matrix, F , describes the set of firewall rules for the network. Firewall rules limit the connectivity by blocking connections between machines. The atomic attacks, A , describe the possible exploits in the model. The initial capabilities, V^0 , are the capabilities present before the attack graph is computed. The final capabilities, V^F , are the capabilities present after the attack graph is computed. The attacker goal, G , is a Boolean expression that states which capabilities the attacker wishes to gain. The edge labels, L , state what atomic attacks were executed in the attack graph.

Let $n = |N|$ and $c = |C|$. The capabilities matrix is a list of capabilities for each machine $i \in N$, $\{V_{i0}, \dots, V_{ic}\}$. The capabilities matrix shows the attacker capabilities in the network according to the following:

$$\sum_{i=1}^n \sum_{j=1}^c V_{ij} = \begin{cases} 1 \rightarrow \text{Attacker has capability } j \text{ on machine } i \\ 0 \rightarrow \text{Attacker does not have capability } j \text{ on machine } i \end{cases}$$

The firewall matrix F is a $N \times N$ matrix describing connectivity between machines in N . A connection is allowed from machine i to machine j if and only if $F(i, j) = 0$.

The initial capabilities V^0 represents all the capabilities the attacker has at the start of evaluation, before executing attacks against machines in N . The final capabilities V^F represents the capabilities the attacker has after the evaluation. If G is NULL then there does not exist any atomic attack template in A whose preconditions are satisfied by V^F . If G is not NULL, then the Boolean expression G is satisfied by V^F .

The list of edge labels takes the form (e, s, t) where e is the name of an atomic attack in A , s is the source machine, t is the target machine and $s, t \in N$. This is for attacks which occur between two hosts, which constitutes the majority of attacks studied. For attacks concerning only one host, this form is still used by setting $s = t$. Attacks concerning more than two hosts were not modeled, but could be accommodated by expanding the host list in the edge label tuple.

An exploit path is a sequence l_1, \dots, l_t , where $l_i \in L$. For all preconditions p in l_1 , the corresponding initial capability V^0_{ij} is equal to 1. For all postconditions p in l_t , the corresponding final capability V^F_{ij} is 1. Thus an exploit path is a sequence of edges from the initial conditions to the final conditions.

4.4 Method

As described in Section 3.1, an attack graph is a collection of exploit paths. Each path is a sequence of atomic attacks and each execution of an atomic attack is an edge in the graph. This method computes the atomic attacks and outputs them in the form of an “attack name, hosts” tuple. The attack name is the label given to the atomic attack. The

hosts attribute describes the specific hosts involved in the attack. Most attacks consist of two hosts, the source and the target. Certain attacks involve only one host, such as privilege escalation, or three hosts, such as man in the middle attacks. The “attack name, hosts” tuple is unique and is only computed once for each possible combination. A visualization tool, currently *dot* [55], composes the output from JESS into an attack graph.

A level based approach based on the model of Ammann *et al.* [23] was used to calculate the “attack name, hosts” tuples for the attack graph. At each level, all atomic attacks which have their preconditions satisfied by the current capabilities of the attacker and state of the network are executed. If the postconditions of the atomic attack alter or add capabilities, these are queued until the next level. Once all possible atomic attacks for the current level have executed, the clean up phase updates and adds all the facts in the queue and increments the level.

The formal model uses matrices to describe the capabilities and firewall rules in the attack graph. Matrices work well formally, but are not as practical with JESS. The matrices are converted into a fact naming structure for the JESS fact base. A fact is only defined for those cells in the matrix that are equal to 1 in the formal model since a value of 0 indicates the absence of that cell in the model. Since facts are only defined for the “present” cells in the matrix, the number of facts to represent the matrix can be much less than the size of the matrix. The size of the matrix is an upper bound on the number of facts.

The capabilities matrix V is represented by the *var* fact which takes the form:

$$(var < host > < type > [< value >] < level >)$$

where *host* is the name of the machine and is equivalent to specifying the row in V , *type* is the capability name and is equivalent to specifying the column in V , *value* is optional for those capabilities which are not Boolean and *level* indicates on which level the capability was obtained. The privilege capability is represented by the special *priv* fact which has the form:

$$(priv < host > < value > < level >)$$

where *host* and *level* are the same as with *var* facts and *value* is 0 for no privilege, 1 for user privilege and 2 for root privilege. With this naming convention, *level* = 0 indicates the initial capabilities V^0 .

The firewall matrix is similarly converted into facts using the *firewall* fact which takes the form:

$$(firewall < host > < src > < tgt > [< port >] < level >)$$

where *host* is the name of the machine that runs the firewall, *src* is the name of the source machine in the attack, *tgt* is the name of the target machine, *port* is an optional port name and *level* is as for the *var* fact.

Each atomic attack is described in a requires/provides format. A rule for an atomic attack has the preconditions on the LHS of the rule and the postconditions in the body of the rule. Additionally, there is a section of the LHS of atomic attack rules that checks the firewall and IDS configurations as given in the network description. These are blocking checks that will prevent the atomic attack from executing if a matching firewall or IDS rule is present in the network description.

Originally, a connectivity matrix as described in [1, 3, 2] was used instead of firewall facts. The matrix is an explicit “allow” method where each allowed connection between hosts must be defined by a connectivity fact. Thus, the connectivity matrix requires $O(n^2)$ facts to implement, where n is the number of hosts. This quickly results in a large number of facts for even moderately sized networks. The fact base size plays an important part in the computation time of the Rete algorithm; larger fact bases lead to longer evaluation time and larger memory use. In practice, most networks are actually configured to implicitly allow connections. That is, connections are allowed by default unless there is a firewall rule blocking the specific connection. For these reasons, a firewall rule methodology is used where there are only facts for explicit firewall rules. In the case where full connectivity between machines is allowed, there would be no firewall facts using the firewall method whereas the connectivity matrix would require $O(n^2)$ facts to represent. There would only be $O(n^2)$ facts for the firewall method when the firewall allows no connectivity to a machine.

In such a case, the machine can just be dropped from the network information file as it cannot be compromised over the network if there is no connectivity to it.

The preconditions and postconditions were originally taken directly from the model presented in [1, 3, 2]. The preconditions and postconditions of that model were specifically crafted to eliminate loops so that it could be used with their model checking tool. The postconditions of an atomic attack would explicitly change the state such that the preconditions would now evaluate to false. This would prevent the atomic attack from being attempted again on that host. This caused an issue in the level-based implementation when computing a tactical style graph, such as Figure 4.4(a), where all possible compromises of a host need to be computed. Consider the state “Trust between host 1 and all” in Figure 4.4(a). This state can be set by executing the FTP attack between three source/target tuples: (A,1), (2,1) and (1,1). However, in the initial state, only the preconditions for the (A,1) tuple are met. Using the original preconditions and postconditions, once the FTP attack from host A against host 1 is executed, no more FTP attacks against host 1 would be attempted. This means the tuples (2,1) and (1,1) would not be computed. The strategic graph in Figure 4.4(b) reflects these limitations of the original preconditions and postconditions. In order to compute the tactical style graph, the preconditions and postconditions have to be reworded.

The current solution is to have two rules defined for each atomic attack. The first rule, the “main” rule, has preconditions and postconditions as defined in [1, 3, 2], with a minor change to the SSHD rule to make it monotonic. To make the SSHD rule monotonic, the variable is set if the SSHD service is enabled in the host’s configuration rather than if the service is currently running. The second rule, called an “edge” rule, rewords the preconditions to look for the consequences of the main rule. For example, if the main rule’s postconditions sets a trust relationship on host y , then the edge rule’s preconditions would look for the presence of a trust relationship on host y . The two rules for the SSHD buffer overflow attack are shown in Table 4.1.

In the SSHD main rule, the precondition that would prevent the loop is the check for the attacker privileges on the target host being less than root. Once this attack, or any other atomic attack that sets the attacker privileges to root, is executed, the main rule’s preconditions would always evaluate to false. This would not be an issue with strategic

Table 4.1: The “main” and “edge” rules for the SSHD buffer overflow attack

SSHD “main” rule
Preconditions
target has sshd service
source attacker privilege \geq user
target attacker privilege $<$ root
\neg firewall(source, target, ssh)
Postconditions
queue update target privilege = root
add tuple (“sshd”, source, target)
SSHD “edge” rule
Preconditions
target has sshd service
\neg tuple (“sshd”, source, target)
source attacker privilege \geq user
target attacker privilege = root
\neg firewall(source, target, ssh)
Postconditions
add tuple (“sshd”, source, target)

graphs as the capability “root on target” has been obtained. For a tactical graph however, the purpose is to determine all the ways the capability “root on target” is obtained. The edge rule determines all the ways which the capability can be obtained by having a precondition that checks for the attacker’s privilege being root on the target machine. In order to prevent the code from going in an infinite loop in the “edge” rule, the list of previously executed tuples for the SSHD attack is checked in the preconditions. This will allow the computation of all possible ways of obtaining the capability “root on target” via the SSHD attack for the tactical view.

The structure of the rules directly correlates to the strategic and tactical views. The exploits executed through “main” rules correspond to the strategic view. Both show that there exists a path to a capability but do not consider all possible paths to the capability. The “edge” rules compute all the additional paths for the tactical view.

Table 4.2: Worst case complexity for Rete, JESS and the attack graph model. Rete complexity is from [4] and JESS complexity is from [5].

Memory	Rete	JESS	Attack Graph
Nodes	$O(P)$	$O(P')$	$O(a)$
Tokens	$O(PW^s)$	$O(P'W'^{s'})$	$O(a(an^2)^s)$
Time	Rete	JESS	Attack Graph
1 Rule	$O(PW^{2s-1})$	$O(P'W'^{s'})$	$O(a(an^2)^s)$
Overall	$O(ZPW^{2s-1})$	$O(ZP'W'^{s'})$	$O(a(an^2)^{s+1})$

P = number of rules in the Rete Network, $P' \leq P$

W = number of facts in the Rete Network, $W' \leq W$

s = average number of patterns per rule, $s' \leq s$

Z = total number of rules executed

a = number of atomic attacks in model

n = number of machines being modeled

4.5 Complexity

The best and worst case complexity for Rete is published in [4]. Only the worst case complexity is considered in Table 4.2 as the worst case complexity gives a better assessment of the scalability of Rete for the attack graph problem. The complexity of the Rete algorithm is affected by both the number of rules in the code, P , and the size of the fact base, W . In general, the time to compute a match for one rule is linear in terms of P and polynomial in terms of W . The memory required for the nodes in the Rete network is linear with respects to P . The memory required for the left and right partial memories, also called tokens, is linear with respect to P and polynomial with respect to W . The average number of logical patterns on the LHS of the rule, s , also affects the complexity. The published worst case complexity for Rete and Jess are summarized in Table 4.2. JESS implements a few optimizations, such as node sharing and hashed left and right memories instead of linked lists, which decreases the memory and computational requirements. This is reflected in the complexity in Table 4.2 by having $P' \leq P$, $W' \leq W$ and $s' \leq s$.

For the attack graph problem, the rules represent the set of atomic attacks A . Let $a = |A|$, then P is $O(a)$ in size. The size of the fact base changes over the execution of the code. Initially, the fact base contains the initial capabilities V^0 and the firewall matrix F . As rules execute, additional capabilities from V are enabled and edge label facts for L are also added. In the worst case, at the end of execution, all capabilities are enabled and L

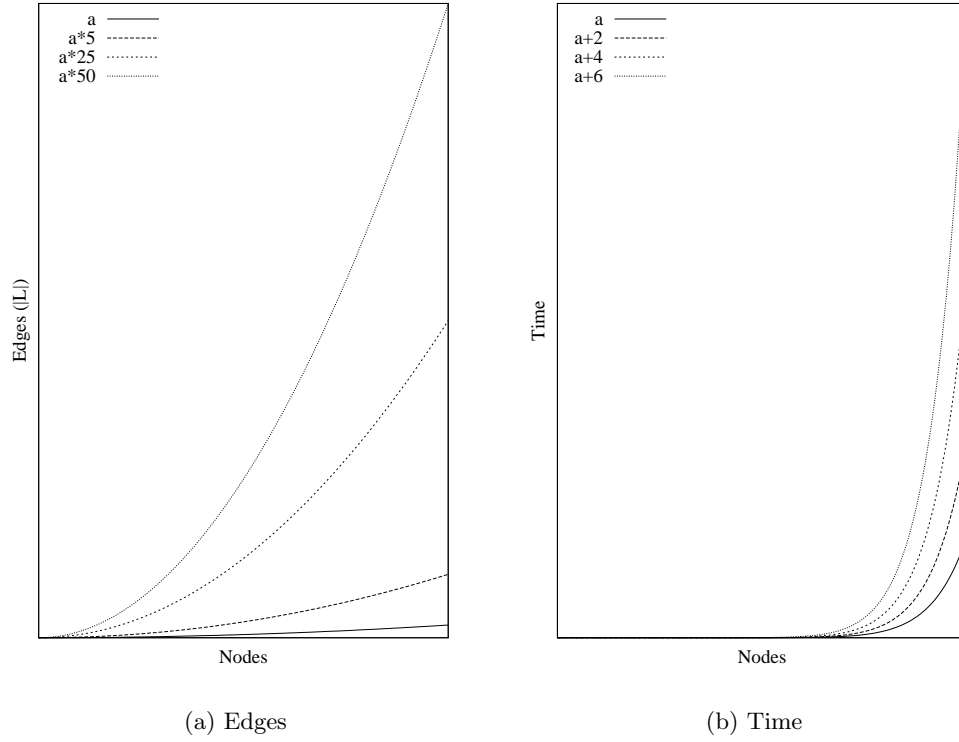


Figure 4.2: The affect of atomic attack set size, a , on the theoretical worst case number of edge labels and runtimes in relation to the number of machines in the network.

is maximum size. So the maximum size of the fact base is $|L| + |V| + |F|$. The size of V is at most cn , the size of F is at most n^2 and the size of L is at most an^2 , as described in Section 4.3. Thus the maximum size of the fact base is $O(an^2)$ as this is the dominating factor. The total number of rules executed, Z , is directly proportional to the number of edge labels calculated as each executed rule adds one edge label. Therefore, Z is $O(an^2)$.

Thus the complexity for the attack graph problem can be expressed in terms of the number of machines modeled, n , and the number of atomic attacks in the model a . This is also summarized in Table 4.2. The number of atomic attacks in particular has a strong effect on the complexity. Consider the graphs in Figure 4.2. While adding more atomic attacks has a moderate effect on the theoretical number of edge labels calculated, as shown in Figure 4.2(a), it has a very strong effect on the computational time as shown in Figure 4.2(b). This is due to the increased computation costs that occur as the fact base size increases with the addition of more edge labels.

4.6 Exploit-Based Model

Individual exploits are used for the atomic attacks in the exploit based model. For proof of concept, four atomic attacks were defined based on the model of Sheyner *et al.* [1, 3, 2]. These attacks are SSHD buffer overflow, FTP writable home directory, RSH login and XTERM buffer overflow. The SSHD buffer overflow allows a remote attacker to gain root privileges on the target host. The FTP writable home directory attack allows the attacker to manipulate the .rhosts file, which establishes a trust relationship between the target host and other hosts. The RSH login attack exploits a trust relationship to give the attacker user level privileges on the target host. The XTERM buffer overflow attack is a privilege escalation attack that allows the attacker to gain root privileges when the attacker originally had user privileges.

4.6.1 Experiments and Results

Several experiments were run using different sample networks to evaluate this method. First, this approach is compared to the model checking approach used in [1, 3, 2]. A second network was also modeled to compare the two methods. These results are presented in Section 4.6.1. Then the scalability of the code for large networks using several common firewall rules is demonstrated in Section 4.6.1

Comparison to Model Checking

The scalability in terms of both memory usage and runtime of this approach was compared to the NuSMV model checker, similar to the one used in [1, 3, 2]. The first experiment compared the results of this method with the results obtained in [1, 3, 2] using the same three node network. The network is shown in Figure 4.3. There are two internal machines and one external attacker machine. There is an IDS monitoring traffic between the attacker and internal machines. This IDS prevents the attacker from launching the RSH login attack from his machine. The goal of the attacker is to achieve root privileges on machine 2.

Figure 4.4 shows the strategic and tactical graph views produced by this method

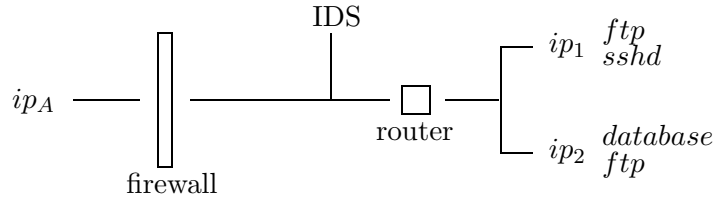
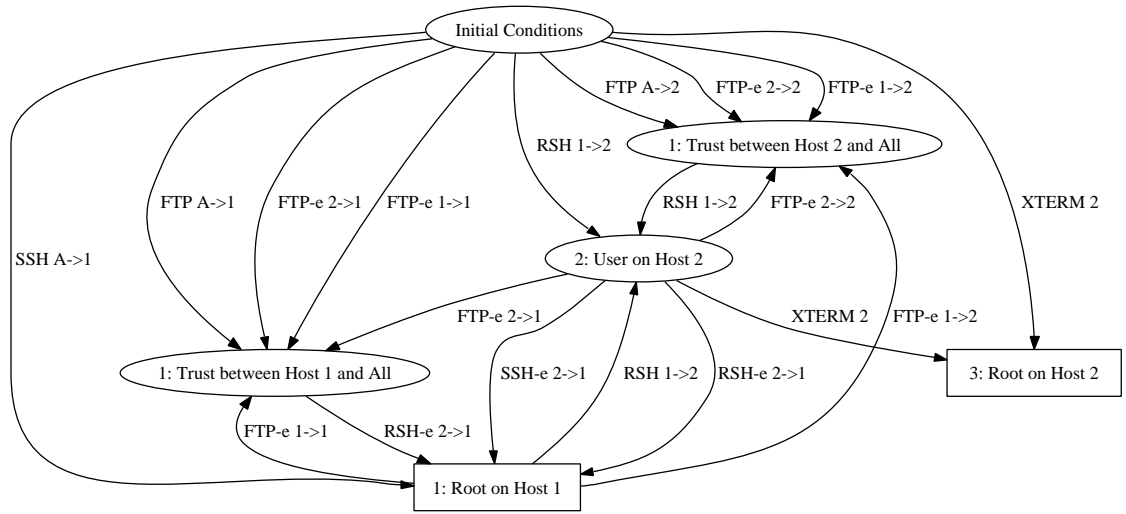
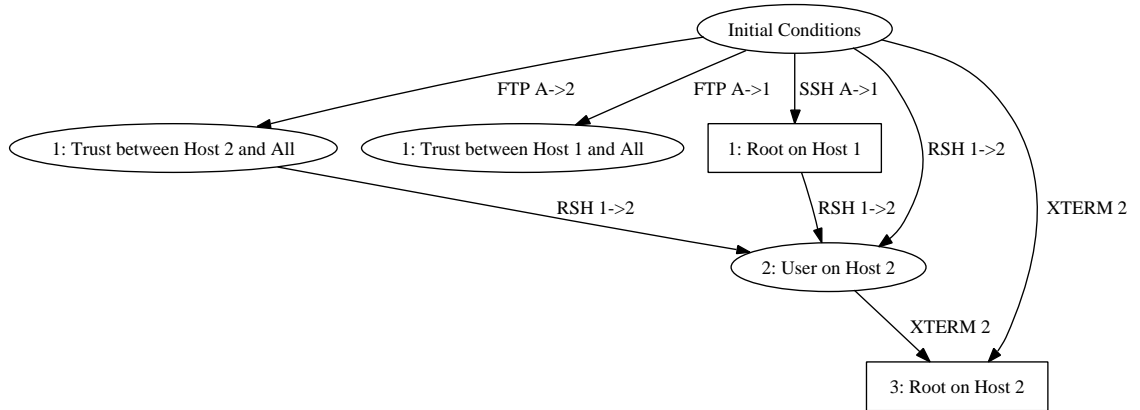


Figure 4.3: The three node sample network.



(a) Tactical



(b) Strategic

Figure 4.4: Attack graph for three node network presented in Sheyner *et al.* and Jha *et al.*. This shows the strategic and tactical graph views.

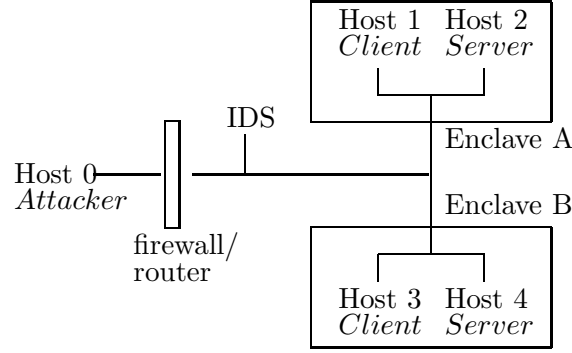


Figure 4.5: The two enclave sample network configuration

on the three node sample network. When compared to the results in [1, 3, 2], this tool computed all the same exploits. It also computed an additional exploit, RSH login attack from host 2 to host 1, that was not included in the results of Sheyner, Jha, *et al.*. While it is a valid attack, it is not particularly interesting because the attacker must have root privileges on host 1 in order to compromise host 2. This method does not do state enumeration like Sheyner, Jha, *et al.* for efficiency reasons. The consequence of not doing state enumeration is that certain redundant exploits, like this, are not flagged as redundant. These redundant exploits can be easily eliminated in post-processing.

The next network that was modeled has two enclaves behind a firewall with the attacker outside the firewall. Each enclave contains a single server and a single client. The network structure and machine names are shown in Figure 4.5. Each server has a web server with a “remote to user” vulnerability and a “user to root” vulnerability. Each client has an OS vulnerability that allows a “remote to root” attack. The attacker initially has no privileges on any computers in the enclave, but has root privileges on his machine. The goal of the attacker is to penetrate the network as far as possible, that is to obtain root on each machine if possible. This is essentially an exploratory graph, but to maintain compatibility with NuSMV, the goal is explicitly defined.

The attacks were implemented using an abstraction of the four previously defined atomic attacks. The client OS vulnerability was mapped to the SSHD buffer overflow attack as both are “remote to root” attack types. The web server “remote to user” vulnerability was mapped to the RSH login attack by setting up default trust relationships in the initial

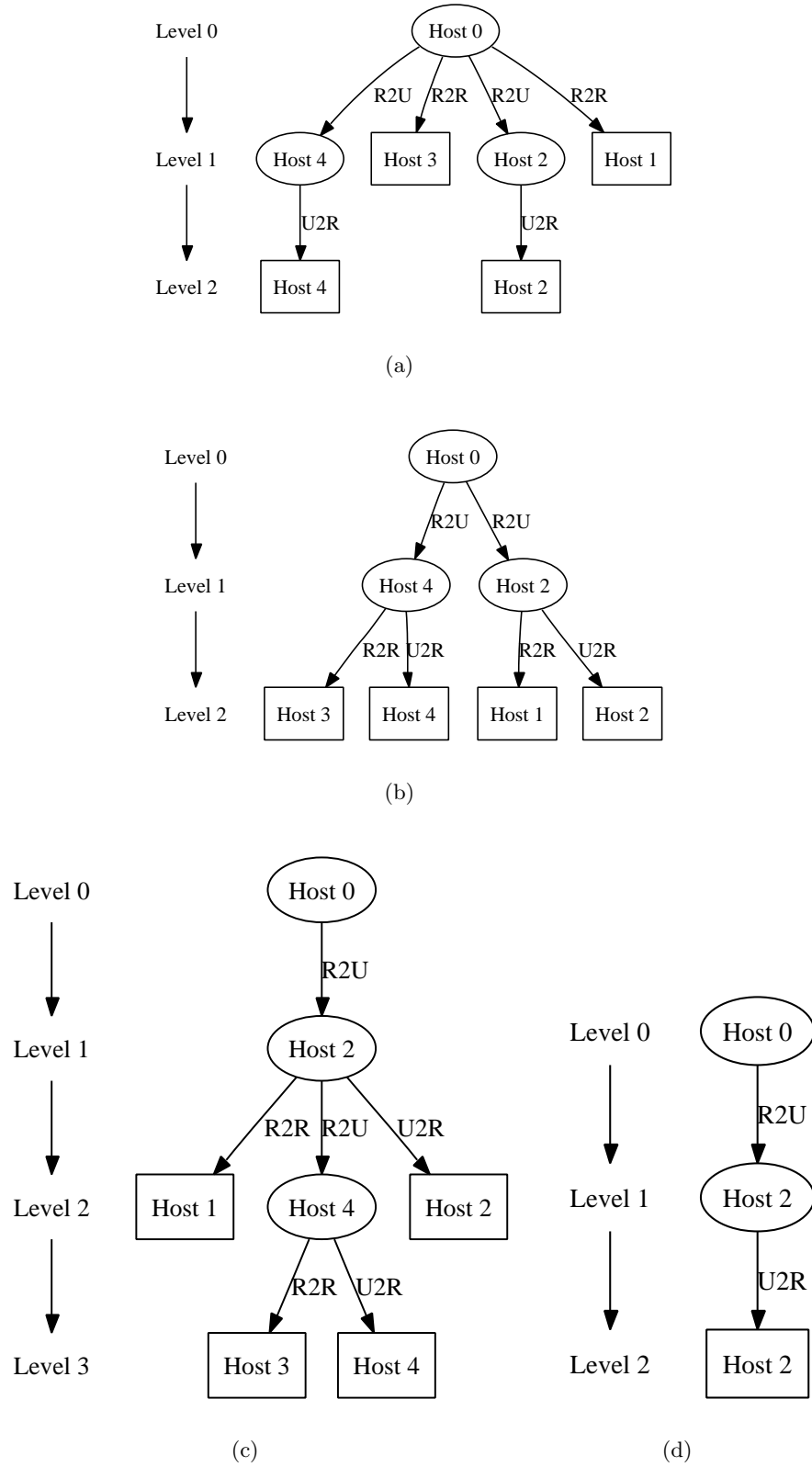


Figure 4.6: Strategic attack graphs for the four enclave scenarios. Graph (a) is scenario 1, (b) is scenario 2, (c) is scenario 3 and (d) is scenario 4. The exploits are remote to root (R2R), remote to user (R2U) and user to root (U2R).

Table 4.3: Enclave sample network: Connectivity for each scenario

Scenario	Host 0	Host 1	Host 2	Host 3	Host 4
1	All	All	All	All	All
2	0,2,4	0,1,2,4	0,1,2,4	0,2,3,4	0,2,3,4
3	0,2	0,1,2,4	0,1,2,4	0,2,3,4	0,2,3,4
4	0,2	0,1,2,4	2	0,2,3,4	4

fact base. The web server “user to root” vulnerability is mapped to the XTERM buffer overflow attack. These abstractions were completely handled by modifying the initial facts.

Four connectivity scenarios were evaluated. In scenario 1, there is full connectivity between all machines on the network. In scenario 2, the attacker can only connect to the servers in the enclaves. Within each enclave, there is full connectivity. Between the enclaves, only the servers can be accessed. In scenario 3, only the server in Enclave A can be accessed by the attacker. The connectivity between enclaves is the same as scenario 2. In scenario 4, again only the Enclave A server can be accessed by the attacker. The connectivity between enclaves follows the following two rules: clients are not allowed to accept connections and servers are not allowed to initiate connections. The connectivity between hosts for each scenario is summarized in Table 4.3.

The scenarios were tested on Pentium 4 2.0 GHz machines with 512MB RAM. The code was able to compute strategic style attack graphs for each scenario in under a second using the default Java VM memory heap size of 64MB. The results for all the scenarios are shown in Figure 4.6. The NuSMV code for scenario 1 originally ran out of memory. After editing the model by hand to remove the connectivity matrix and unlikely scenarios, the NuSMV code ran in under 3 seconds. For the remaining scenarios, the NuSMV code ran out of memory even after optimizing the model by hand. This approach appears to be significantly more scalable in terms of memory from these tests. The runtime scalability could not be fully compared due to the NuSMV code running out of memory before finishing.

Scalability with Large Networks

In order to test how well this method scales to large networks, hundreds of flat topology networks like that shown in Figure 4.7 were generated. All of the networks have the same five server segment, Segment A, as described in Table 4.4. There is also a second

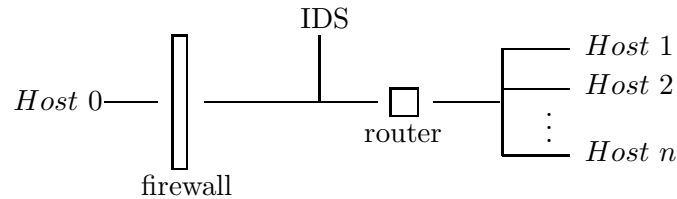


Figure 4.7: The flat network topology.

Table 4.4: Flat network: Segment A server names and vulnerabilities

Host	Vulnerabilities
0	n/a (attacker)
1	ssh, rsh
2	ftp, rsh
3	ftp, rsh, xterm
4	ssh, rsh, xterm
5	ssh, ftp, rsh, xterm

segment, Segment B, that contains both clients and servers. Segment B has a variable number of hosts, up to 90,000 hosts, with each host being randomly assigned vulnerabilities. In all the networks, the attacker is external to the network and cannot use the RSH login attack from his machine, as with the sample network in [1, 3, 2].

Three firewall scenarios were tested to determine the effect of the firewall rules on runtime and memory requirements. The first scenario, no firewall, has no firewall rules so there is full connectivity between all hosts. The second scenario, “crunchy-chewy” (CC), has a firewall that prevents the attacker’s original host from initiating a connection to any of the hosts in Segment B. Thus the attacker must first compromise a server in Segment A before he can compromise a host in Segment B. The third scenario, DMZ, has separated Segment A and Segment B. The attacker can only initiate connections to the servers in Segment A. The servers in Segment A cannot initiate connections to the hosts in Segment B, but they can initiate connections to other servers in Segment A. Thus, the attacker is limited to compromising the servers in Segment A. Additionally, this means that the size of the attack graph remains the same regardless of how many hosts are in Segment B.

The tests were run on a Pentium 4 3.2 GHz machine with 1GB of RAM. The Java VM was invoked with a maximum heap size of 768MB. For the “no firewall” and “CC” scenarios, an out of memory error occurred for networks with more than 1500 hosts. The

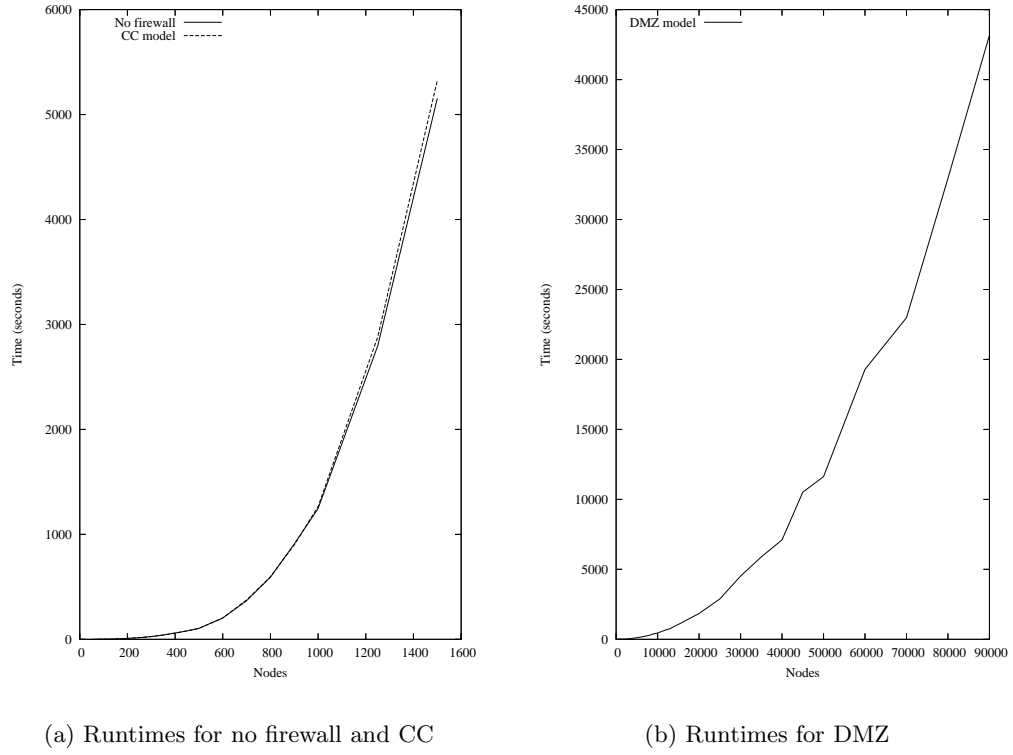


Figure 4.8: Runtimes for the flat network topology.

runtimes are shown in Figure 4.8(a). For the “DMZ” scenario, the impact of a large network when the number of attacks remained static was tested. There were 43 attacks in the DMZ graph. It was able to run up to 90,000 node networks without running out of memory, but the runtime was quite long at nearly 12 hours. Figure 4.8(b) shows the runtimes for all the networks tested.

As described in Section 4.4, this method uses a level-based approach to compute the edges. This scalability testing showed that the majority of the runtime is spent in the clean up phase between levels. Most of the time in this phase is spent activating the new facts. The larger the fact base is, the longer this phase takes to complete.

In summary, this method scales better in terms of memory and computational time than the NuSMV model checker. It is also able to compute graphs for networks containing over a thousand nodes in Section 4.6.1. This is equivalent to several class C networks.

4.7 Conclusion

This chapter has presented a method to generate attack graphs that is more scalable than model checking methods, has customizable visualization and uses a scripting language to eliminate extensive hand coding. This method is more scalable in terms of memory use and computation time than the NuSMV model checking method, as shown by the tests done in Section 4.6.1.

The generation of exploit paths is separated from the visualization of the attack graph to allow for greater flexibility and interoperability with other visualization methods such as [27]. This method also supports exploit path filtering to create strategic and tactical style attack graphs. Thus the visualization can be customized based on the intended use of attack graphs.

By using the CLIPS [58] derived scripting language provided by JESS, adding new atomic attack templates does not require extensive hand coding. This language also makes the code portable to other expert systems which support the CLIPS syntax. The code has been ported to CLIPS, although the performance with JESS was better.

The tests have shown that this method is scalable to more realistic networks than the small networks often used in previous works. It can scale to several class C networks using the exploit-based approach. The abstract class model presented in Chapter 5 gains additional scalability over the exploit-based approach.

Chapter 5

Abstract Class Model

5.1 Introduction

As was shown in Section 4.5, the number of rules in the model greatly affects the scalability. Both memory and runtime are dominated by the number of atomic attacks in the model and the number of machines in the network. This chapter details a new model which reduces both of these factors. Abstraction is used to reduce the number of rules in the model by developing abstract classes of rules. Clustering is used to reduce the number of machines in the network by clustering identical machines into one “meta” machine.

In order to achieve greater scalability, the abstract class model was developed. The process of abstraction is essentially grouping multiple similar exploit based rules into one abstract rule. Exploit based rules are considered *similar* if they have preconditions that only vary by variable name and have postconditions that result in the same capabilities for the attacker. For example, consider the exploit based attacks SSH and IIS. For the preconditions, both operate over a single network port and require a vulnerable network service on that port. For the postconditions, both result in the attacker having root privilege on the machine. Both could be abstracted into an abstract rule for obtaining root privileges by exploiting a vulnerable network service. A side-by-side comparison of SSH, IIS and the abstract rule are shown in Table 5.1.

The difficulty with abstraction is in determining the abstract classes. Towards this end, previous works in vulnerability classification were investigated. While this provided

Table 5.1: Comparison of the RSH and IIS exploit based attack rules with the abstract attack rule.

SSH	IIS	Abstract
Main Preconditions		
$var(t, ssh)$	$var(t, iis)$	$var(t, pe_noauth)$
$priv(s) \geq user$	$priv(s) \geq user$	$priv(s) \geq user$
$priv(t) < root$	$priv(t) < root$	$priv(t) < root$
$connect(s, t, ssh)$	$connect(s, t, web)$	$connect(s, t, pe_noauth)$
Edge Preconditions		
$var(t, ssh)$	$var(t, iis)$	$var(t, pe_noauth)$
$priv(t) = root$	$priv(t) = root$	$priv(t) = root$
$priv(s) \geq user$	$priv(s) \geq user$	$priv(s) \geq user$
$\neg edge(s, t, ssh)$	$\neg edge(s, t, iis)$	$\neg edge(s, t, r2r_noauth)$
$connect(s, t, ssh)$	$connect(s, t, web)$	$connect(s, t, pe_noauth)$
Postconditions		
$priv(t) = root$	$priv(t) = root$	$priv(t) = root$

insight into the issue of classification, many of the prior works focused on classifying how a vulnerability is enabled and less on the consequences (“postconditions”) of the vulnerability. The problem domain was restricted to methods by which an attacker could escalate his privileges on a system. The primary effects of abstraction are to reduce the size of the atomic attack set and the size of the capabilities set.

In addition to abstraction, clustering was also developed in this model. Clustering is the process of grouping identical machines into one “meta” machine. Identical machines will have identical vulnerabilities and can be compromised in an identical fashion. Including all the machines within the model thus replicates work. Once the paths to and from a single machine in the group has been computed, the paths to and from all the machines in the group are known. By default, the Rete algorithm will compute the paths for all the machines as it does not know that the machines are identical. By clustering the machines into one “meta” machine, the computation is only done once by Rete. The effect of clustering is then to reduce the size of the machine set and decrease the repetitive work done by Rete in the exploit based model. Clustering could be done with the exploit based model as well, but it works synergistically with the abstract class model to provide much greater scalability.

Section 5.2 details the theoretical effects of reducing the atomic attack set size, the capabilities set size and the machine set size. Section 5.3 details the prior works into

vulnerability classification. Section 5.4 presents the details of how the abstract model was developed, how the model works and how clustering fits in with the abstract model. Section 5.5 provides results of several experiments with the abstract model, both with and without clustering, and compares them to the results of the exploit based model.

5.2 Motivation

In Section 4.5, the complexity of the attack graph model was detailed. The worst case complexity of the attack graph model is influenced by the number of atomic attacks and the number of machines in the model. This is summarized in Table 4.2. To get better scalability with the attack graph model, focusing on reducing the number of atomic attacks and the number of machines in the model yields the most productive results. To reduce the number of atomic attacks, an abstract attack model has been developed. To reduce the number of machines, clustering of identical machines in the same network segment was implemented.

A smaller atomic attack size set leads to better scalability as was shown in Figure 4.2. A moderate decrease in the size of the atomic attack set lead to a moderate decrease in the number of edge labels, but a dramatic decrease in the theoretical runtime. Abstracting the atomic attacks causes a decrease in the size of the atomic attack set by abstracting many exploit based rules to a single abstract rule. Another effect of abstraction is to reduce the size of the capability set. Usually each exploit based rule will require a unique capability such as the presence of a particular vulnerability. When multiple exploit based rules are abstracted, these multiple capabilities are replaced with a single abstract capability. Thus the size of the capability set, c , also decreases under the abstract model. From Section 4.5, the size of the fact base is $O(cn + n^2 + an^2)$ and the size of the fact base is one factor in both the runtime and memory requirements of the Rete algorithm. While an^2 is the dominating factor, decreasing c also affects the size of the fact base. Thus the abstract model achieves greater scalability by decreasing both a , the size of the atomic attack set, and c , the size of the capabilities set. Figure 5.1 shows how reducing both a and c by two affects the worst case runtime. Even a modest reduction in the sizes of the atomic attack set and capabilities

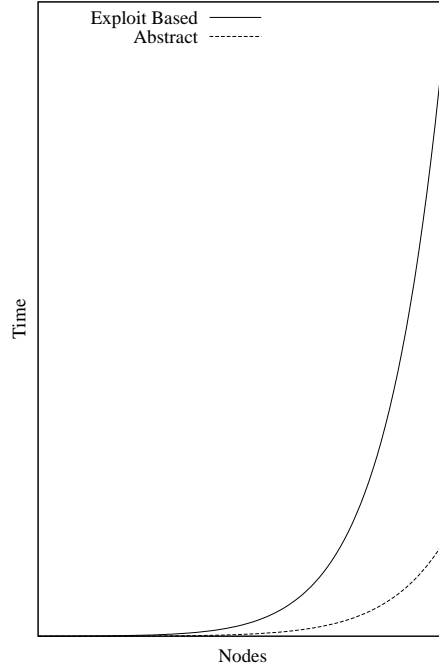


Figure 5.1: The theoretical worst case runtime for an exploit based model versus an abstract model. The exploit model has 9 rules and 11 capabilities with an average of 6 patterns per rule. The abstract model has 7 rules and 9 capabilities with an average of 6 patterns per rule.

set has a large effect on the theoretical runtime.

Besides the scalability advantages, abstract atomic attacks also simplifies the amount of code required to add a new exploit to the model. If the exploit falls under a current abstract atomic attack, it can be added to the model by adding the exploit to abstract mapping to the model. The model also still allows for exploit based atomic attacks if they do not fall under a predefined abstract atomic attack. This does sacrifice some performance, but allows for flexibility to model more unique exploits.

Clustering identical machines in the same network segment reduces the number of machines, n , in the model. Identical machines are identified by finding machines which have the same capabilities and the same connectivity. In this model, most of the capabilities are binary, indicating either the presence or absence of that capability. The privilege capability is the only exception to this, having three possibilities. So the total number of possible combinations of capabilities is $3 * 2^c$. If all machines had the same connectivity, then clustering could reduce the size of the machine set to at best $3 * 2^c$ when the number of

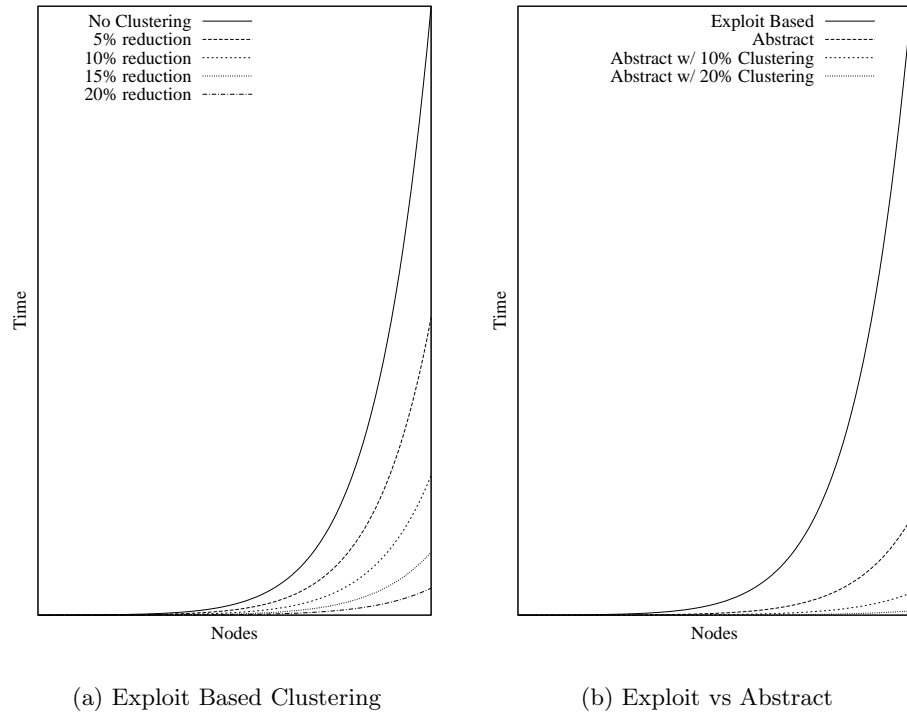


Figure 5.2: The effects of clustering on theoretical runtimes. The clustering is shown by its effectiveness in reducing the size of the machine set. In (a), the clustering is done on the exploit based model from Figure 5.1. In (b), the clustering is done on the abstract model from Figure 5.1 with the unclustered exploit based model runtime included for comparison.

machines, n , is greater than $3 * 2^c$. Since very few networks will have all machines with the same connectivity, the clusters must also consider the connectivity of the machines. Only machines on the same logical network segment, that is those machines with identical firewall rules on all routers in the network, should be included in a cluster. Let n^s be the number of distinct network segments. Then at best, clustering can reduce the number of machines to $3 * 2^c * n^s$. Even when the number of machines is less than $3 * 2^c * n^s$, clustering can still be effective in reducing n if there are identical machines in the network. At worst, no clusters are found and the number of machines remains n . Figure 5.2(a) shows the effects that clustering has on the exploit based model used in Figure 5.1 when clustering reduces the number of machines by 5-20%. Figure 5.2(b) shows the effects of reducing the number of machines by 10% and 20% for the abstract model from Figure 5.1. Clustering decreases the runtimes for both the exploit based and abstract models. However, this effect is more pronounced for the abstract model.

Clustering is most effective in larger networks where n is greater than $3 * 2^c * n^s$ or in networks where identical machines exist in some network segments. Clustering dovetails well with abstraction since abstraction reduces the size of the capability set which reduces c and thus reduces the maximum number of clusters. Clustering also dovetails well with abstraction as each method independently reduces the runtime.

5.3 Previous Works

The RISOS study [59] focused on operating system vulnerabilities. It developed seven general classes of vulnerabilities: incomplete parameter validation, inconsistent parameter validation, implicit sharing of privileged or confidential data, asynchronous validation or inadequate serialization, inadequate identification or authentication or authorization, violable prohibition or limit and exploitable logic error. It also proposed techniques to avoid flaws and methods to detect flaws in the incomplete parameter validation class.

The PA study [6] focused on operating system and program vulnerabilities. It used a technique called *pattern-directed protection evaluation*. This was a technique to evaluate existing systems using feature extraction and comparison of features. They had

Table 5.2: The classification scheme for the PA study [6].

1. Improper protection domain initialization and enforcement
 - (a) Improper choice of initial protection domain
 - (b) Improper isolation of implementation detail
 - (c) Improper change
 - (d) Improper naming
 - (e) Improper deallocation or deletion
2. Improper validation
3. Improper synchronization
 - (a) Improper indivisibility
 - (b) Improper sequencing
4. Improper choice of operand or operation

a hierarchy of error patterns from abstract to concrete errors. They derived generalized patterns by using generic names, using abstract features and deleting unnecessary details. They developed the classification scheme detailed in Table 5.2.

Landwehr, et al. [7] asked three basic questions about a vulnerability: how did it enter the system, when did it enter the system and where in the system did it manifest. The flaws were classified according to these three questions into flaws by genesis (how), flaws by time of introduction (when) and flaws by location (where). These classifications schemes are summarized in Table 5.3.

Aslam, et al. [60] and Aslam’s thesis [61] presents a classification of UNIX security faults. The scheme is composed of two parts: coding faults and emergent faults. Coding faults result from synchronization errors or condition validation errors. Synchronization errors include a timing window between two operations and improper serialization of operations. Condition validation errors include a missing condition, an incorrectly specified condition, a missing predicate in the condition or a missing or misspecified check for limits, check for access rights, check for valid input or check for origin of subject. Emergent faults result from configuration errors or environmental issues. Configuration errors include putting the program or utility in the wrong location, installing the program or utility with improper parameters or having a secondary object that is installed with incorrect permis-

Table 5.3: The classification schemes presented in Landwehr, et al.[7].

Flaws by Genesis			
Intentional			
Malicious			
Trojan Horse			
Non-replicating			
Replicating			
Trapdoor			
Logic/Time Bomb			
Non-malicious			
Covert Channel			
Storage			
Timing			
Other			
Inadvertent			
Validation error (incomplete/inconsistent)			
Domain error (object reuse, residuals, exposed representation)			
Serialization/aliasing			
Identification/Authentication inadequate			
Boundary condition violation (resource exhaustion, violable constraint)			
Other exploitable logic error			
Flaws by Time of Introduction			
During development			
Requirement/Specification/Design			
Source code			
During maintenance			
During operation			
Flaws by Location			
Software			
Operating System			
System initialization			
Memory management			
Process management/scheduling			
Device management (including I/O, networking)			
File management			
Identification/authentication			
Other/unknown			
Support			
Privileged utilities			
Unprivileged utilities			
Application			
Hardware			

sions. A series of questions are presented to classify a fault into one of these categories.

Lindqvist and Jonsson [8] developed a classification scheme for a set of attacks used in an experimental setup. The experiment limited the scope of their attacks to attacks done by authorized users of a system. They focused on a concept called dimension, which is the attribute used as a basis for classification, and on developing a classification from the owner's point of view. The dimensions of interest they present from this point of view are intrusion techniques and intrusion results. Intrusion techniques are how the vulnerability is exploited and intrusion results are the immediate results of the vulnerability being exploited. Their classification scheme is summarized in Table 5.4.

The vulnerability analysis project at University of California, Davis [62, 9] classifies vulnerabilities with respects to the security policy. There are a set of characteristics required for a vulnerability to exist. The similarity of any two vulnerabilities can be given by set intersection of their characteristics sets. Negating any of the characteristics in the set for a vulnerability will prevent that vulnerability and any other vulnerability which also has that characteristic in its set. The work also considers the point of view and level of abstraction. It aims to develop a classification scheme that is consistent across all points of view and levels of abstraction. The work into protocol analysis [9] also considers the symptoms as part of the classification scheme. A symptom is defined as the violation of the policy. An exploit is defined as a pairing of symptoms with vulnerabilities. The work presents a grammar in Extended Backus-Naur Form (EBNF) for network protocol vulnerabilities as summarized in Table 5.5.

The seminal works on vulnerability classification are the RISOS study [59], the PA study [6], Landwehr, et al.'s taxonomy of program flaws [7] and Aslam, et al.'s taxonomy of Unix faults [60, 61]. These works focused on classifying how vulnerabilities occur and as such are less relevant to the attack graph work. They did however provide insight into methodologies to use to develop abstract classes, particularly the PA study with its concept of an error hierarchy and generalized classes.

The work of Lindqvist and Jonsson [8] separates how vulnerabilities occur from the consequences of the vulnerability. This is similar to the attack graph concept of preconditions and postconditions, except that with attack graphs, the preconditions are the

Table 5.4: The classification schemes presented in Lindqvist and Jonsson [8].

Intrusion Techniques	
Bypassing intended controls	<ul style="list-style-type: none"> Password attacks <ul style="list-style-type: none"> Capture Guessing Spoofing privileged programs Utilizing weak authentication
Active misuse of resources	<ul style="list-style-type: none"> Exploiting inadvertent write permissions Resource exhaustion
Passive misuse of resources	<ul style="list-style-type: none"> Manual browsing Automated searching <ul style="list-style-type: none"> Using a personal tool Using a publicly available tool
Intrusion Results	
Exposure	<ul style="list-style-type: none"> Disclosure of confidential information <ul style="list-style-type: none"> Only user information disclosed System and user information disclosed Service to unauthorized entities <ul style="list-style-type: none"> Access as an ordinary user account Access as a special system account Access as client root Access as server root
Denial of Service	<ul style="list-style-type: none"> Selective <ul style="list-style-type: none"> Affects a single user at a time Affects a group of users Unselective <ul style="list-style-type: none"> Affects all users of system Transmitted <ul style="list-style-type: none"> Affects users of other systems
Erroneous Output	<ul style="list-style-type: none"> Selective <ul style="list-style-type: none"> Affects a single user at a time Affects a group of users Unselective <ul style="list-style-type: none"> Affects all users of system Transmitted <ul style="list-style-type: none"> Affects users of other systems

Table 5.5: The classification grammar presented in [9].

Characteristic Grammar

```

<characteristic> ::= MISCONFIGURATION | MISSPECIFICATION | SOFTWARE FLAW |
                    HARDWARE FLAW | AUTHENTICATION FLAW | AUTHORIZATION FLAW |
                    NONCE FLAW | STATE MACHINE FLAW

```

Vulnerability Grammar

```

<vulnerability> ::= MISCONFIGURATION+ | <implementation>+ | <design>+
<implementation> ::= SOFTWARE FLAW | HARDWARE FLAW
<design>           ::= MISSPECIFICATION | STATE MACHINE FLAW | <forgery>
<forgery>         ::= AUTHENTICATION FLAW | AUTHORIZATION FLAW | NONCE FLAW

```

Symptom Grammar

```

<symptom>           ::= DIVERT <resource> | DISABLE <resource>
<resource>          ::= <user resource> | <host resource> | <network resource>
<user resource>     ::= USER CREDENTIALS | USER DATA
<host resource>     ::= HOST CREDENTIALS | HOST DATA | HOST SERVICE
<network resource>  ::= CONNECTION | BANDWIDTH

<sniffing>          ::= DIVERT USER DATA
<impersonating>     ::= DIVERT USER CREDENTIALS | DIVERT HOST CREDENTIALS
<connection hijacking> ::= DIVERT CONNECTION and DIVERT HOST CREDENTIALS
<man in the middle> ::= <connection hijacking>+
<flood>             ::= DISABLE BANDWIDTH
<denial of service> ::= DISABLE HOST SERVICE | <man in the middle> | <flood>

```

Exploit Grammar

```

<exploit> ::= <vulnerability>+ <symptom>+

```


requirements for an attack to occur, not the requirements for a vulnerability. Their intrusion results scope goes beyond the scope of the abstract class model, including denial of service and erroneous output, while the attack graph focuses on privilege escalation and information leaks that can lead to a privilege escalation. Their work focuses solely on developing a classification scheme while the abstract model focuses on developing a scheme and using that scheme to achieve more scalability when evaluating a network. The vulnerability analysis project at University of California, Davis [62, 9] shares some similarities with the attack graph work, but it also focuses more on developing a classification grammar than on the composition of attacks across the entire network.

5.4 Model

To develop a working abstract model, privilege escalation was chosen as the subject to be modeled. Most serious attacks utilize privilege escalation at some stage. Other factors in network security such as denial of service, quality of service, spoofing and social engineering were not modeled, but they could be included in future models. Methods of privilege escalation modeled were program vulnerabilities such as buffer overflows, weak or default passwords, information leaks such as the password hash, trust exploitations and network information leaks. Also included in the model was the concept of firewall rules and attacks that could alter or delete firewall rules.

The atomic attacks were developed based on the roll of the machine. The machine entities in this model were servers, clients, routers and mixed machines. Servers are machines which have services that accept connections from other machines such as a web server. Servers have vulnerabilities which are exploited by the attacker connecting to the server. Clients are machines which make connections to other machines. Clients are exploited by the attacker tricking the user into connecting to a machine controlled by the attacker, such as a compromised web server. Routers are machines which contain the firewall rules. A router has vulnerabilities that are also exploited by the attacker connecting to the router, however the consequences of a router compromise also includes altering or deleting firewall rules. Mixed machines contain elements of two or more entities, such as a

client/server machine. There are also local vulnerabilities for both server and client entities which can be exploited only once the attacker has gained user privileges on a machine.

The preconditions and postconditions for the abstract atomic attacks were derived by looking at the exploit based model, considering prior works in vulnerability classifications, considering methods by which privileges can be escalated and looking at plugins for the Nessus vulnerability scanner [63]. For example, consider the *ssh-buffer-overflow* and *iis-buffer-overflow* attacks in the exploit based model as shown in Table 5.1. The difference between the two rules is the name of the capability the attacker must have on the target machine and the name of the port on which the targeted service is located. The other preconditions concerning the attacker privileges on the source and target are identical and the postcondition is also identical. These two rules can be abstracted into a single rule by using an abstract name for the capability and port as shown in Table 5.1. The abstract capability *pe_noauth* substitutes for the capabilities *ssh* and *iis*. The preprocessing would map any *ssh* or *iis* capabilities to *pe_noauth*. Likewise, the ports *ssh* and *web* are abstracted to the port name *pe_noauth*. The abstract rule is called *r2r-noauth* in the model, so when this is used in place of *ssh* and *iis* in the edge rule preconditions.

The derived classification scheme for the abstract model separates the methods used by each entity from the attacker results. The classification scheme for each entity is presented in Table 5.6 and the scheme for the attacker results is in Table 5.7.

The four attacker capabilities are summarized in Table 5.8. The attacker has a privilege level on each machine. This is an ordered set of privileges, $\text{none} < \text{user} < \text{root}$. A common goal of an attacker is to gain root privilege on a machine. The trust capability means the attacker has succeeded in establishing a trust relationship with that machine or the machine had a trust relationship in the initial capabilities of the network. If the machine has a trust login service, the attacker can use the trust relationship to gain user privileges on the machine. The two types of information disclosure modeled are system information and network information. System information is usually encrypted passwords that the attacker can run a dictionary attack on to recover the original password. It can also include less common methods of forging authentication such as cryptanalysis or improper clean-up of memory. Network information can include data such as a port scan of a machine that the

Table 5.6: Classification scheme for the server, client, router and local entities.

Server entity - Methods over the network against servers

Invalid authorization/authentication

Privilege escalation

Authorized user

Unauthorized user

Using trust relationship

Using system information

Authorized user

Unauthorized user

Using password guessing

Authorized user

Unauthorized user

Using network information

Authorized user

Unauthorized user

Invalid trust establishment

Information disclosure

System information

Authorized user

Unauthorized user

Network information

Authorized user

Unauthorized user

Client entity - Methods utilizing client trojans

Invalid authorization/authentication

Privilege escalation

Authorized user

Unauthorized user

Information disclosure

System information

Authorized user

Unauthorized user

Network information

Authorized user

Unauthorized user

Router entity - Methods over the network against routers

Invalid authorization/authentication

Privilege escalation

Bypass firewall rules

Local entity - Methods using binaries on a host

Invalid authorization/authentication

Privilege escalation

Using system information

Using password guessing

Information disclosure

System information

Network information

Table 5.7: Classification scheme for the attacker results.

Unauthorized privileges
 Access as user
 Access as root
 Information disclosure
 System information
 Network information
 Invalid trust relationship
 Bypass firewall rules

Table 5.8: Capabilities relating to what the attacker has gained.

privilege	Privilege level of the attacker on the host. One of: none, user, root
trust	Attacker has established a trust relationship on the host, host trusts all other hosts for remote trust login
info_system	Information relating to passwords or authentication
info_network	Information about the network, such as port scans

attacker could not scan from outside of the network.

For server machine entities, there are sixteen rules as summarized in Table 5.9 and twenty one capabilities as summarized in Table 5.10. The rules cover a variety of methods for escalating privilege by exploiting a networked service, gathering system information and using that to guess passwords and having weak, default or brute force guessable passwords. The direct exploitation of a vulnerable network service requires that the firewall rules allow an attacker controlled machine to connect to the target and that the target have a vulnerable network service. Information gathering involves a two-stage attack where the attacker first exploits a network service to gain information about the system or the network, but not user or root privileges. The system information can contain information such as password hashes which the attacker can then attempt to decode and use to log into the system as either a user or root, depending on the password recovered. The network information can contain port scans and similar network related information that could reveal servers with vulnerable services. If the attacker can connect to these servers, he could then target them with attacks on those services to obtain user or root privileges. Unset, default or easily guessed passwords is another method by which an attacker can obtain user or root privileges on a system on which he can connect to the login service.

For client entities, there are seven rules summarized in Table 5.11 and seven ca-

Table 5.9: Server related abstract rules in the abstract attack graph model.

Rule	Description
r2r-noauth	Allows an attacker with less than root privileges to obtain root privileges
r2r-auth	Allows an attacker with user privileges to obtain root privileges
r2u	Allows an attacker to obtain user privileges
trust-est	Allows writing to files used in trusted logins
trust-exp	User login service that uses trust for authentication
info-system-noauth	Leaks system information to attacker with less than root privileges
info-system-auth	Leaks system information to attacker with user privilege
r2r-info-system	Root login service where the attacker has obtained the root password via a system information leak
r2u-info-system	Attacker logs into remote login with user password obtained from system information
r2r-weak	Root login when the system has a default, unset or brute force guessable root password
r2u-weak	User login where the user has a a default, unset or brute force guessable root password
info-network-noauth	Leaks network information to attacker with less than root privileges
info-network-auth	Leaks network information to attacker with user privilege
r2r-noauth-info	Gives root privileges to attacker with less than root privileges using network information
r2r-auth-info	Gives root privileges to attacker with user privilege using network information
r2u-info-network	Gives user privileges to attacker with no privileges using network information

Table 5.10: Server capabilities in the abstract attack graph model.

Variable	Description
pe_noauth	Vulnerable service that gives root privileges to attacker with less than root privileges
pe_auth	Vulnerable service that gives root privileges to attacker with user privileges
pe_user	Vulnerable service that gives user privileges to attacker with no privileges
write_noauth	Vulnerable service that allows attacker with no privileges to write to filesystem
trust_login	Remote login service that uses trust relationships for authentication and gives user privileges to authenticated clients
leak_noauth	Vulnerable service that leaks system information to attacker with less than root privileges
leak_auth	Vulnerable service that leaks system information to attacker with user privileges
login	Remote login service using passwords for authentication and gives either root or user privileges to authenticated clients
root_login	Root login allowed via remote login service
crackable_root_password	Root password can be deducted from system information
crackable_user_password	User password can be deducted from system information
weak_root_password	Root password is default, unset or guessable with brute force
weak_user_password	User password is default, unset or guessable with brute force
leak_network_noauth	Vulnerable service that leaks network info to attacker with less than root privileges
leak_network_auth	Vulnerable service that leaks network info to attacker with user privileges
pe_noauth_info	Vulnerable service that gives root privileges to attacker with less than root privileges, requires network information
pe_auth_info	Vulnerable service that gives root privileges to attacker with user privileges, requires network information
pe_user_info	Vulnerable service that gives user privileges to attacker with no privileges, requires network information

Table 5.11: Client related abstract rules in the abstract attack graph model.

Rule	Description
client-r2r-noauth	Gives root privileges to attacker with less than root privileges
client-r2r-auth	Gives root privileges to attacker with user privilege
client-r2u	Gives user privileges to attacker with no privileges
client-info-system-noauth	Leaks system information to attacker with less than root privileges
client-info-system-auth	Leaks system information to attacker with user privilege
client-info-network-noauth	Leaks network information to attacker with less than root privileges
client-info-network-auth	Leaks network information to attacker with user privilege

Table 5.12: Client capabilities in the abstract attack graph model.

Variable	Description
client_pe_noauth	Vulnerable client program that gives root privilege to attacker with less than privileges
client_pe_auth	Vulnerable client program that gives root privilege to attacker with user privilege
client_pe_user	Vulnerable client program that gives user privilege to attacker with no privileges
client_leak_noauth	Vulnerable client program that gives system information to attacker with less than privileges
client_leak_auth	Vulnerable client program that gives system information to attacker with user privilege
client_leak_network_noauth	Vulnerable client program that gives network information to attacker with less than privileges
client_leak_network_auth	Vulnerable client program that gives network information to attack with user privilege

pabilities summarized in Table 5.12. These rules involve tricking a client into connecting to a machine controlled by the attacker, such as injecting malicious code into a website that the client then visits with a vulnerable web browser. The malicious code can result in the attacker getting user or root privileges on the client system or it can gather system or network information. If the client machine is a mixed entity with a remote login service, the system information can be used much in the same fashion as with servers to retrieve passwords and log on to the system. Network information is also used in a similar fashion to server entities to find new machines to attack.

For router entities, there are two rules summarized in Table 5.13 and two capabilities summarized in Table 5.14. The router rules encapsulate two ways by which an attacker

Table 5.13: Router related abstract rules in the abstract attack graph model.

Rule	Description
router-wan-to-admin	Gives attacker root privileges on router and disables all firewall rules for router
router-firewall-disable	Disables all firewall rules for router

Table 5.14: Router capabilities in the abstract attack graph model.

Variable	Description
wan_admin	Vulnerable router that gives root privileges via WAN port to attacker with no privileges and disables firewall for that router
firewall_disable	Vulnerable router that disables firewall for router to attacker with no privileges

can disable firewalls. The *router-wan-to-admin* attack involves gaining root privileges on the router, which would allow the attacker to do anything he desires including deleting firewall rules. For simplicity, it is assumed that any router on which the attacker gains root privileges has all its firewall rules deleted. The *router-firewall-disable* rule covers any methods by which an attacker could bypass, delete or alter firewall rules without having privileges on the router. For example, the router might have a vulnerability in its protocol stack that would allow an attacker to bypass host name based firewall rules with a specially crafted packet. If such a vulnerability exists in the router, this atomic attack assumes the attacker can use the vulnerability to bypass all firewall rules in the router, so the postcondition for the atomic attack is to delete all firewall rules for that router from the model.

There are also five local rules summarized in Table 5.15 and four local variables summarized in Table 5.16. The local rules cover methods by which an attacker with user privileges on a system might escalate his privileges to root, gather system information or gather network information. For example, a vulnerable binary might allow an attacker to escalate his privilege to root by giving it unexpected data. There could also be a local login binary that allows an attacker who is able to deduce the root password either from system information or because it is a weak password to escalate his privileges. Local system information leaks can come from vulnerable binaries, unprotected memory, world readable permissions on the password file, unprotected temporary files and so on. Network information leaks can come from tools such as network traffic sniffers and port scans that an attacker with user privileges may be able to run on the machine.

Table 5.15: Local binary related abstract rules in the abstract attack graph model.

Rule	Description
priv-esc	Allows an attacker to escalate user privilege to root privilege
info-system-local	Leaks system information to attacker with user privilege
r2r-info-system-local	Login that allows an attacker to escalate from user to root with a root password obtained from a system information leak
r2r-weak-local	Root login when the system has a default, unset or brute force guessable root password
info-network-local	Leaks network information to attacker with user privilege

Table 5.16: Local capabilities in the abstract attack graph model.

Variable	Description
pe_local	Vulnerable binary that gives root privileges to attacker with user privileges
leak_local	Vulnerable binary that leaks system information to attacker with user privileges
local_login	Local login binary exists that allows root login to attacker with user privileges
leak_network_local	Vulnerable binary that leaks network info to attacker with user privileges

In total, thirty abstract rules were defined using thirty four abstract capabilities. Also included in the model was the privilege capability for a total of thirty five capabilities in the model. The complete preconditions and postconditions for the abstract rules are given in Appendix D. The appendix also notes examples of exploit based rules that are related to each abstract rule, such as *r2r-noauth* being an abstraction of *ssh-buffer-overflow* and *iis-buffer-overflow*.

5.4.1 Preprocessing

The preprocessing for the abstract model takes the exploit based network information file and translates it into the abstract model. This involves having a mapping from the exploit based capability names to the abstract capability names and a mapping from the exploit based port names to the abstract port names. For many of the mappings, this is a one to one translation from exploit based to abstract. However, there are some abstract capabilities which require multiple exploit based capabilities. An example is the abstract capability *write-noauth* which is an abstraction of three exploit based capabilities, *ftp*, *fshell* and *wdir*. All three exploit based capabilities need to be present on a machine for the *write-*

noauth abstract capability to be set in the network information file. The pre-processing has a special mapping to indicate all abstract capabilities like *write-noauth* which require more than one exploit based capability to be set. It checks for the presence of all required capabilities on the machine before setting the abstract capabilities in the network information file.

The port name mapping for firewall rules also is mostly a one to one mapping from exploit based to abstract port name. The mapping to abstract port names also filters out any firewall rules unrelated to machines in the network as they are unnecessary. There is also a special circumstance with some abstract port names. For example, the abstract port name *pe_noauth* covers the exploit based ports *ssh* and *web*. If the exploit based firewall rules have a rule blocking access on the *ssh* port, but not one for the *web* port, the abstract firewall rules should not block connections to *pe_noauth*. A special mapping is used for those abstract port names that cover more than one exploit based port name. The abstract port name firewall rule is only added to the network information if there is a firewall rule blocking every exploit based port for the machines in question.

The preprocessing also handles any assumptions that exist in the exploit based model. For example, in the exploit based model based of Sheyner's works [1, 3, 2], it is assumed that all machines have the *rsh-trust-login* service enabled. In the abstract model, no such assumption is made and instead the capability *trust-login* is used to indicate the presence of a trust based login service on a machine. When preprocessing network information files that were used for that exploit based model, the preprocessing needs to set *trust-login* for each machine to handle the assumption in the exploit based model.

5.4.2 Clustering

The clustering algorithm first groups machines into preliminary clusters by their capabilities. It then considers in turn each preliminary cluster and subdivides the clusters by network segment. Grouping by capabilities is accomplished by creating a string that encodes the capabilities of the machine and using that string as a key into a hash table. The string is of length c with each character in the string encoding the state of the corresponding capability. The machine set is scanned sequentially and each machine with the

same capability string is put into an array of machine names in the same hash table slot.

After all the machines are put into the hash table, each slot with a set of machines in it is examined. If there is only one machine in that slot, there is no cluster and it is printed out to the network information file as is. If there is more than one machine, the first machine's firewall rules are compared to the other machines in the hash table slot. In order for them to be considered on the same network segment, two machines being compared must use the same routers and have the same rules on each router with respect to their connectivity to other machines. Routers are not allowed to be clustered. Any router in a preliminary cluster is removed from the preliminary cluster and the printed out to the network information file as is. After all the machines on the same network segment as the first machine in the preliminary cluster have been identified and moved to a final cluster, the process repeats with the remaining machines in the preliminary cluster until all network segments have been identified.

Once the final clusters have been created, a unique name is assigned to each cluster. The capabilities for the cluster are output to the network information file with this unique name. The firewall rules are scanned and any rule which refers to a machine in the cluster is substituted with the unique cluster name. Since a cluster will have duplicate firewall rules due to the fact that clustering is done based off having identical firewall rules, all duplicate firewall rules are removed before outputting the firewall rules to the network information file.

5.4.3 Postprocessing

The postprocessing for the most part creates human readable reports and graphs from the abstract model. It looks at the exploits computed by the abstract model and translates the required capabilities for those exploits back to the exploit based names for human readability. Since the preprocessing has special circumstances in handling port names for abstract ports that cover more than one exploit based port, the postprocessing must handle this. To return to the previous example from the preprocessing section, if *ssh* is firewalled in the exploit based model, but *web* is not firewalled, the abstract model will allow connections to *pe_noauth*. The postprocessing must account for this fact when

translating any *r2r* abstract attacks into human readable reports. The postprocessing will filter out *ssh* from the report as it would not be allowed with the firewall rules as long as the firewall was not compromised by either *router-wan-to-admin* or *router-firewall-disable* in an earlier round than the *r2r* attack.

5.5 Experiments

The experiments show how the abstract model and clustering performs on sample networks. A comparison between a limited exploit based model and abstract model is presented in Section 5.5.1.

5.5.1 Comparison to Exploit Based Model

For this experiment, an exploit based model consisting of nine atomic attacks was compared to an abstract model containing seven atomic attacks that are abstractions of the exploit based model. The exploit based attacks used were *sshd-buffer-overflow*, *ftp-rhosts*, *remote-login*, *xterm-overflow*, *iis-buffer-overflow*, *squid-port-scan*, *licq-user-shell*, *client-browser-trojan* and *at-overflow*. These attacks are detailed in Appendix C. The abstract atomic attacks used were *r2r-noauth*, *priv-esc*, *trust-est*, *trust-exp*, *info-network-noauth*, *r2u-info-network* and *client-r2u*. These attacks are detailed in Appendix D.

A five subnet network was used to compare the exploit based model, the abstract model and the abstract model with clustering. The basic network topology is given in Figure 5.3. There is a DMZ, four subnets and the attacker controlled machine *host₀*. The firewalls allow connections from *host₀* to any host in the DMZ, but does not allow *host₀* to connect to any machines in any of the four subnets. The machines in the subnets are allowed to connect to any machine in the DMZ or *host₀*. The machines in each subnet *i* are allowed to connect to other machines in subnet *i*, but not machines in other subnets. With respects to the actual firewalls, *host₀* and machines in the DMZ can connect to *fw₀* but no other firewalls and machines in subnet *i* can connect to firewall *fw_i*.

For testing, eighty one data files were generated, with the number of hosts per network segment ranging from 1 to 500. Vulnerabilities were randomly assigned to each

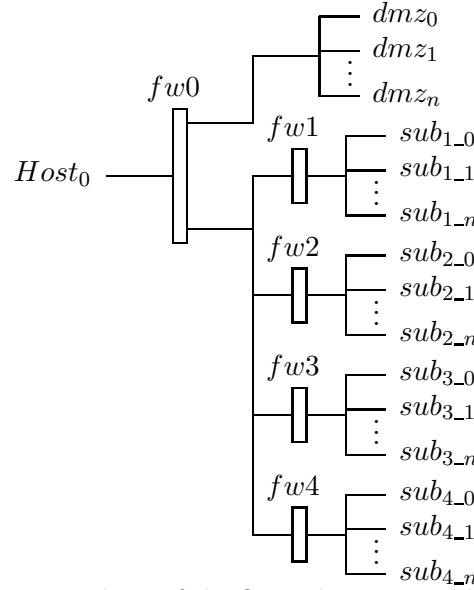


Figure 5.3: The basic topology of the five subnet network used to compare the exploit based, abstract and abstract with clustering models. The DMZ subnet contains the machines that the attacker can directly connect to from his initial machine.

machine with differing probabilities of setting each vulnerability. The *ssh* vulnerability had a 20% probability of being set. The *iis*, *ftp*, *wdir* and *fshell* independently had a 50% chance of being set. Additionally, if *ftp* is set, there was a 50% probability of also setting *wdir* and *fshell*. The *xterm* vulnerability had a 15% probability of being set. The *squid* and *licq* vulnerabilities had a 10% chance of being set. The *browser* vulnerability had a 75% probability of being set. The *at* vulnerability had a 25% chance of being set.

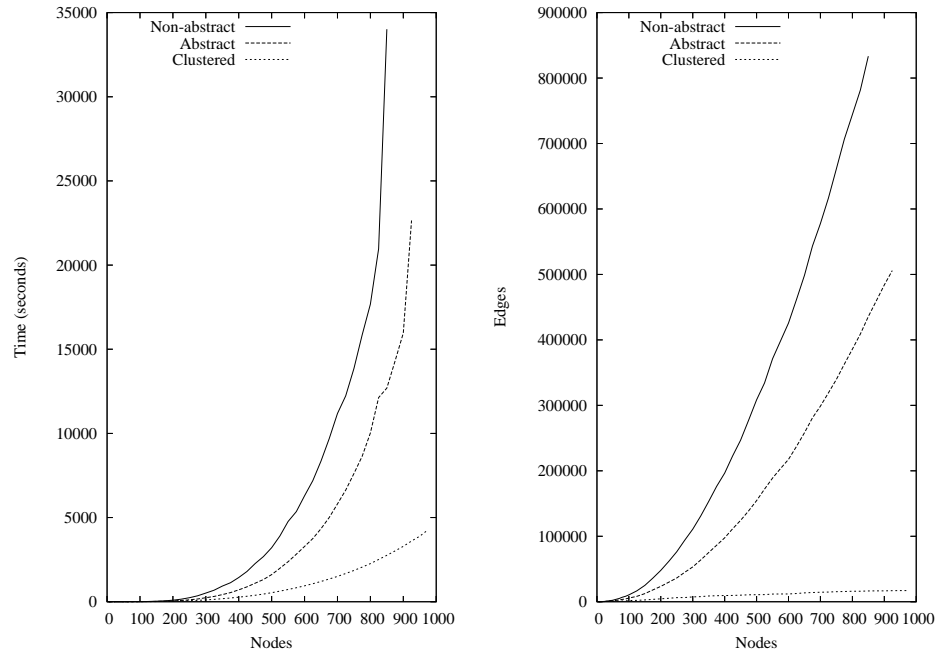
Once these exploit based data sets were generated, they were converted to abstract data sets by running the preprocessing. The preprocessing time was recorded and included in the results. The clustering algorithm was then run on the abstract data sets to create data sets for the abstract model with clustering. Again, the time to run the clustering algorithm was recorded and included in the results.

The tests were run on a Pentium 4 3.2 GHz machine with 1GB of RAM. The clustering algorithm was implemented in Perl and slowed down to 1% CPU utilization on the data set which contained 350 hosts per subnet, for a total of 1756 hosts. This is likely due to the way Perl handles running out of physical memory. For running JESS, the Java VM was invoked with a maximum heap size of 896MB. The exploit based model ran out of memory on the data set containing 856 total hosts. The abstract model ran out of memory

on the data set containing 931 total hosts. The abstract model with clustering was able to run on all the data sets that were converted.

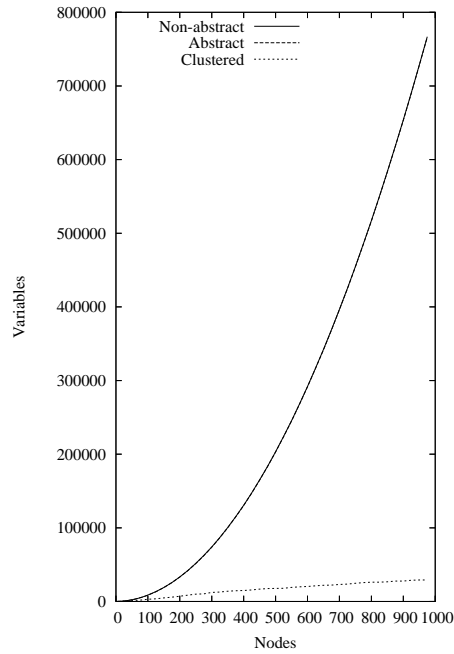
The results are presented in Figure 5.4. For all data sets, the abstract model outperformed the exploit based model. Since the atomic attack set was reduced by only two rules in this comparison model, it shows the strong affect the atomic attack set size has on performance. On average, the abstract model took half the time to run as the exploit based model. The abstract model was also able to run on larger data sets due to both the reduction in the number of variables in the network information and to the memory savings in Rete that results from having fewer atomic attacks. The reduction in the number of initial variables from abstraction was actually rather small, ranging from 0.08% to 7.69%. This is likely due to the fact that the random assignment of vulnerabilities would not often assign two exploit based vulnerabilities that could be abstracted to a single abstract vulnerability. The better utilization of memory by the abstract model most likely comes from effect the smaller atomic attack set had on the working memory requirements of Rete. From Table 4.2, the memory requirements is $O(a)$ for the number of nodes and $O(a(an^2)^s)$ for the number of tokens for partial memory. By decreasing a with the abstract model, this lowers the memory requirements of Rete for attack graph calculations.

The clustering algorithm combined with the abstract model had the most drastic decrease in the runtime, number of initial variables and number of edges calculated as shown in Figure 5.4. Only the smallest data set with 1 host in each network segment and 11 hosts total had no clusters. For the remaining files, several clusters were found, ranging in size from a two node cluster up to a sixty node cluster. Figure 5.5(a) shows the maximum, minimum and average cluster size for the data sets. Figure 5.5(b) shows the percent reduction in the number of hosts for the data sets. As the number of machines in the data sets increased, so did the percentage of machines that could be clustered. From the network containing 181 hosts and beyond, clustering was able to reduce the number of machines by 50% or more. This yielded enormous runtime benefits with JESS. The majority of the time shown in Figure 5.4(a) was for running the Perl program to find and label the clusters. Once the clusters data sets were created, JESS took one minute or less to process the data sets, even for the network that originally contained 1706 hosts. This



(a) Runtime

(b) Edges



(c) Variables

Figure 5.4: A comparison of the runtimes, the number of edges calculated and the number of initial variables for the non-abstract exploit-based model, the abstract model and the abstract model with clustering. Runtimes include preprocessing time.

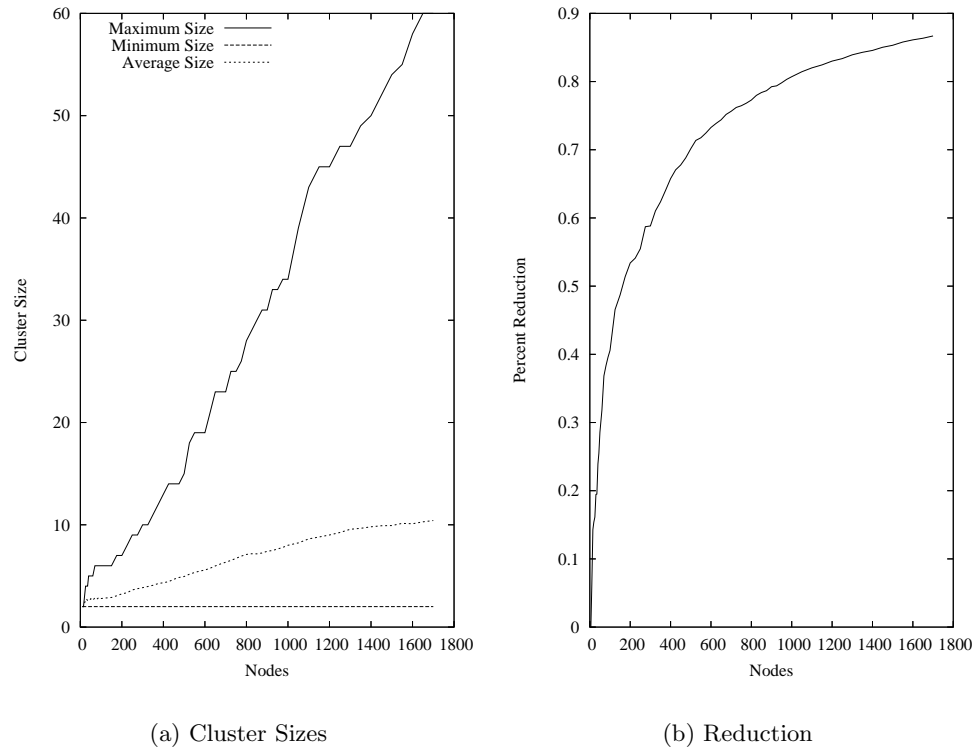


Figure 5.5: Results for the clustering algorithm. In a), the maximum, minimum and average cluster sizes are shown. In b), the percent reduction in the number of hosts is shown.

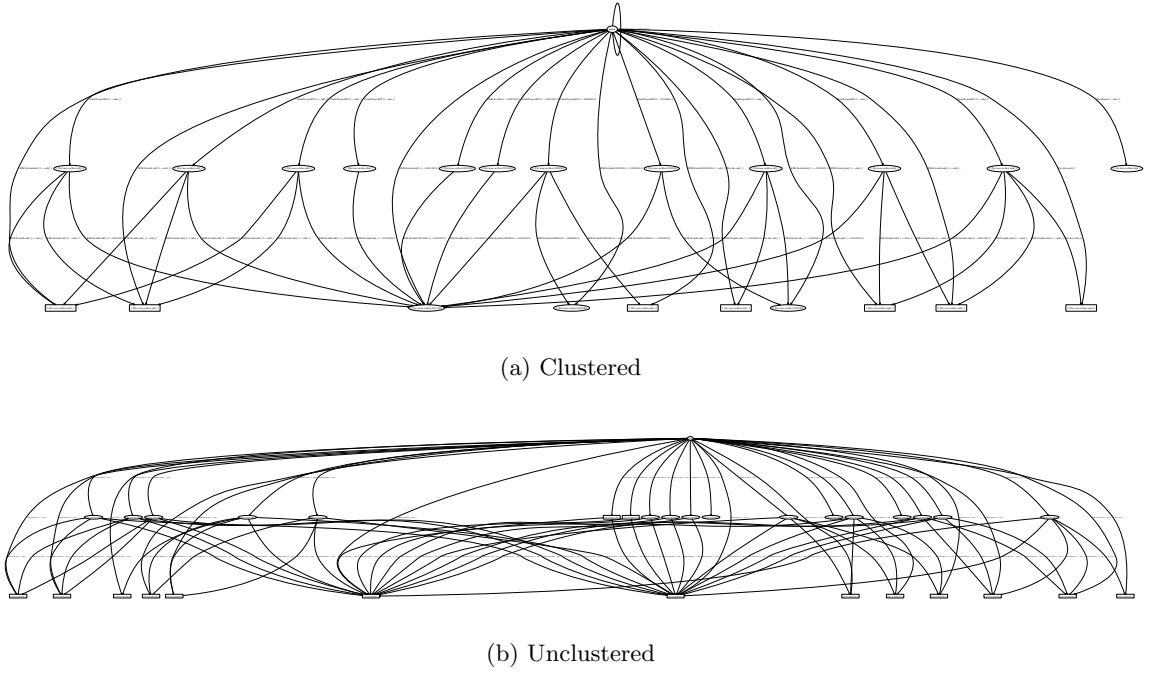


Figure 5.6: Comparison of the visual complex of the attack graph using clustering and not using clustering. This is the attack graph for the network with 3 hosts per subnet.

dramatic increase in runtime is not unexpected given the large percentage of hosts that were able to be clustered. The Rete algorithm runtime complexity for the attack graph problem is $O(a(an^2)^{s+1})$, from Table 4.2. The reduction in the number of hosts seen for the data set with 856 machines originally was 78%, with the number of machines reduced to 185 machines. With the theoretical runtime complexity, one would expect to see a 99.9% reduction in the Rete runtime for that data set. The actual reduction in runtime was 99.7%.

Additionally, the visual complexity of the resulting attack graph is reduced by the clustering method. As shown in Figure 5.4(b), the number of attack edges computed with clustering is dramatically less than the number of attack edges computed with the unclustered abstract model. For the larger networks, clustering produced 2% of the edges produced by the unclustered abstract method. Even for the network with three hosts per subnet, clustering resulted in half the number of edges compared to unclustered. The resulting clustered and unclustered attack graphs are shown in Figure 5.6. Clustering also reduces the number of nodes in the attack graph. In Figure 5.6, the reduction is minor

as there are only a few possible clusters. The unclustered graph has 32 nodes while the clustered graph has 23 nodes. For the larger network with 170 hosts per subnet, the clustered graph has 278 nodes while the unclustered graph has 1472 nodes. The number of nodes for the clustered method is about 20% of the number of nodes for the unclustered methods for this larger network.

5.6 Conclusion

The motivation behind the abstract model is to improve scalability and reduce coding complexity. The abstract model has better scalability in terms of runtime and memory usage than the exploit based model. This is true even when the abstract model contains only a few less rules than the exploit based model. The abstract model is also able to analyze larger networks than the exploit based model. The clustering algorithm combined with the abstract model has a drastic effect on the runtime and memory. The time to identify and calculate the clusters dominates the time to analyze the resulting clustered network. The reduction in the computation time of the attack graph is as high as 99% when clustering is used. The combination of clustering and abstraction makes it feasible to compute attack graphs for larger networks than the exploit based model.

Coding complexity is also reduced by the abstract model. To add new attacks to the model, there are two options: code a new rule or add the attack to the mapping for an existing rule. If the abstract rules cover the problem domain, most new attacks should fall under the classification of one of the abstract rules. Thus, it is expected that most new attacks would simply have to be added to the mapping for an existing rule. This greatly reduces both the skill and time required to add a new attack to the model.

Chapter 6

Analysis of Attack Graphs using Evolutionary Computation

6.1 Introduction

Presenting an attack graph to an administrator is often not helpful as the graph is highly complex. Even with intricate visualization methods designed to reduce visual complexity, it is desirable to perform some sort of analysis on the attack graph to give the user some direction to take. The type of analysis is dependent on the use of attack graphs. The author believes that the two most common uses of attack graphs are for network design and patch management. With these uses, the desired outcome of the analysis is a more secure network. For forensics and intrusion detection correlation, the analysis would be to predict further paths the attacker could have taken in the network and provide a list of items to investigate further such as a path the attacker may have taken. For intrusion response, the analysis should predict the next steps of the attacker and chose the most reasonable responses to block the attacker.

For each of these points of view, different methods of analysis are required. This section focuses on the network design and patch management point of view. See Chapter 7 for future work that could perform the analyses for the other points of view. The goal of the analysis for network design and patch management is to come up with a set of hardening

measures that will prevent the attacker from reaching his goals, if such a set of measures is possible. In the model of this thesis, a hardening measure is an action which removes a precondition from an atomic attack. Three possible hardening measures are patching a vulnerability, firewalling a port and placing an IDS sensor such that it would detect the exploit. Every hardening measure has a cost associated with it. Typically these costs are determined by the policy of the network. For example, if the policy states the web server on machine A must be publicly acceptable, the hardening measure “firewall port 80 to machine A” should have an infinite cost associated with it so that hardening measure is not used. The costs can also reflect the reality of the network, such as the availability of patches or ease of deploying patches. Even if two networks have the same attack graph, they may have different sets of hardening measures depending on their policies. The analysis algorithm must allow for this site specific customization.

Several prior works [24, 1, 2] have shown that determining the minimum set of hardening measures is a reduction of the set cover problem and thus NP. An approximation method must be used to determine the minimum set of hardening measures. This work explores the use of evolutionary computation to derive the set of hardening measures. Refer to Section 2.4.1 for a general introduction to evolutionary computation.

6.2 Related Works

Philips and Swiler [24] determined a set of low cost paths by taking the logarithm of the weights and computing Dijkstra’s shortest path to the goal node. They also could compute sets of low cost paths using bicriteria shortest path algorithms or Naor and Brutlag’s algorithm. They also looked at the problem of maximally decreasing the probability that the attacker will succeed given a set of possible defenses. They showed that to determine the set of cost effective defenses is NP-hard as it is a reduction of the set cover problem. Additionally, they showed that determining the minimum sensor placement to maximize coverage over low cost paths is also NP-hard. They suggest an iterative technique where the administrator alters the configuration file then recomputes the shortest paths to see if the alterations have improved the security of the network. They also propose a simulation

tool to show the administrator the paths an attacker is most likely to take from a given starting point.

Sheyner, et al. [1, 2, 3] performed several analyses, primarily to find the minimal critical set of attacks and the minimum critical set of countermeasures. The minimal critical set of attacks is the set of attacks that cannot be removed from the attack graph without preventing the goal node from being reached. Finding the minimal critical set of attacks essentially finds all vulnerabilities that need to be patched. The minimum critical set of countermeasures finds the minimum number of patches or other countermeasures that need to be applied to prevent the attacker from reaching the goal. They show that both of these problems are NP-complete and provide greedy approximation algorithms. Their algorithm to find the minimal critical set runs on the order of $O(mn)$ where m is the number of states and edges and n is the number of attacks. It recursively calls the function `isCritical(C)`, which runs in $O(m)$, adding attacks to the set C until a critical set is found. The attacks added to the set can be chosen with respects to a cost metric. The `isCritical` function determines if the goal is still reachable if all the edges related to the attacks in C are removed. Similarly, their algorithm to compute the minimum critical set of countermeasures chooses countermeasures until the goal is no longer reachable. The countermeasures can also be chosen with respects to a cost metric.

Noel, et al. [26, 28] perform recursive algebraic substitution on the attack graph starting at the goal node. The goal node is substituted with the conjunction of its preconditions. The preconditions are then substituted with the disjunction of all exploits that yield that precondition. This substitution continues recursively up the tree until the initial conditions are reached. The resulting expression is then rewritten in canonical conjunctive normal form. The expression is thus reduced to a collection of “maxterms”, where each maxterm is a possible assignment of initial conditions that will prevent the attacker from reaching the goal. They arrange the maxterms according to the partial order of their non-negated conditions and they chose the minimal maxterms. In other words, a maxterm is chosen if its non-negated conditions are a subset of the non-negated conditions of the other maxterms. If costs are assigned to hardening conditions, the selection can also compute the total cost of a maxterm and use that as a part of the selection criteria. Their example

uses the network from Section 4.6.1. The minimal maxterms selected by their algorithm correspond to either “prevent ftp(0,2), prevent ftp(0,1) and prevent ssh(0,1)” or “prevent ftp(0,2) and prevent ftp(1,2)”

Ammann, et al. [23] mark their nodes during the attack graph generation phase to make analysis easier. Each node contains a list of all attacks which can make that node true. Three methods of analysis are given: findMinimal, findAll and findShort. The findMinimal algorithm computes the minimal critical attack set. The findAll algorithm finds all paths to the given node set. The findShort algorithm modifies findMinimal to consider the number of steps from the initial state, choosing attacks which have fewer steps. The runtime of their algorithms are quadratic in terms of the number of exploits. They use the network from Section 4.6.1 as an example. The findAll algorithm returns three paths:

1. ssh(0,1), ftp(1,2), rsh(1,2), xterm(2,2)
2. ftp(0,2), rsh(0,2), xterm(2,2)
3. ftp(0,1), rsh(0,1), ftp(1,2), rsh(1,2), xterm(2,2)

6.3 Approach

Previous analysis methods have ranged from the “try and test” approach using a simulation tool [24], greedy approximation algorithms [1, 2, 3, 23] and expressing the graph as a logical expression of the initial capabilities [26, 28]. The “try and test” method requires extensive interaction by the user. The greedy approximation algorithms of [1, 2, 3] have runtimes that are exponential in terms of the size of the network and their methods do not scale well to larger networks. Ammann’s [23] method is quadratic in terms of the number of attacks, but it only finds the set of minimal critical attacks, not the lowest cost set of hardening measures. The logical expression algorithm of [26, 28] only looks at patching the initial vulnerabilities, not at applying other hardening measures. None of the prior methods consider the policy of the network when determining a set of hardening measures. The genetic algorithm method uses multiple hardening procedures and considers the policy of the network in determining the plan of action.

A genetic algorithm was used to determine a set of hardening measures that maximize the security of the network while minimizing the cost. The security of the network is measured by how well the goal nodes are disconnected from the graph by the hardening measures. This work focuses on deriving hardening measures during the course of patch management or network design. Therefore, the possible hardening measures are:

- Install patch x against an initial condition (vulnerability). This removes a node from the graph corresponding to that vulnerability.
- Firewall connection between hosts a and b on a specified port. This will remove attack edges that depend on network connectivity.
- Place an IDS sensor for attack x occurring between hosts a and b . This will mark edges in the graph as being “watched”. The attack can still succeed but will be detected by the IDS.

Each of these measures has a cost associated with it, as defined by the policy of the network. If the policy does not define a cost, then default costs are assigned depending on the purpose for analyzing the attack graph. If the purpose is to aid the design of a new network, firewalling connections is preferred and gets a lower cost. If the purpose is to determine which patches to install on an existing system, then installing patches is preferred and has a lower cost.

The chromosome for the genetic algorithm is the concatenation of three bitmaps corresponding to each type of hardening measure. The first bitmap contains one bit per node in the initial capabilities of the network. If bit i is 1, this means the patch for the vulnerability in the initial capability node i will be installed. If the bit is 0, no patch will be installed. The second bitmap corresponds to the edges in the attack graph. If bit i is 1, then the network connection required for edge i ’s attack will be firewalled. If it is 0, then there will be no firewall for that connection. The third bitmap also corresponds to the edges in the attack graph. If bit i is 1, then an IDS rule that will detect the attack between the two hosts in edge i will be enabled. If the bit is 0, no IDS rule for that attack will be enabled.

Some hardening measures will be prevented by the policy of the network or by the nature of an attack. The policy may prohibit the installation of certain patches or it may specify connectivity requirements for a host that would make it impossible to use certain firewall rules. Additionally, not all attacks occur over the network. Some attacks are local to a specific host. For these attacks, a firewall would be ineffective. An IDS would have to be host-based to detect local attacks as well. There are two approaches to handle this. Either the cost associated with such a prohibited action can be set to infinity or the bits corresponding to such actions can be removed from the bitmap or ignored. Both approaches will be explored.

In order to evaluate how the hardening measures indicated in the chromosome affect the security of the network, the fitness function must first apply the actions and then see how the actions affect the goal node(s). For patches, the node corresponding to the patched vulnerability is removed from the graph along with all of its outgoing and incoming edges. For firewall rules, the edge corresponding to the attack that depends on that connectivity is removed. The process of removing edges will affect the in-degree on other nodes in the attack graph. If the in-degree becomes 0, in other words all incoming edges for that node have been removed, then that node is also removed. For IDS rules, the edge is marked as “watched”. Each node contains a bitmap of the incoming edges that are watched. The node to which the edge points sets the corresponding bit in the bitmap to 1. If all incoming edges to a node are watched, the node marks all outgoing edges as watched.

The fitness function determines the sum of the costs for all enabled hardening measures and looks at the goal nodes to determine how well the measures improved the security of the system. The sum of the costs is the total cost for applying those hardening measures. The improvement to the security of the system is the benefit. The ideal result is to have all goal nodes removed from the attack graph. A goal node will only be removed if all its incoming edges are removed. If not all edges can be removed, say due to policy restrictions, then next best result is to have all remaining edges watched. Thus the benefit to the system is based on how many incoming edges have been removed from the goal nodes and how many of the remaining edges are watched. Removing an edge is given a higher weight than watching an edge so that removing all edges will have a higher benefit than

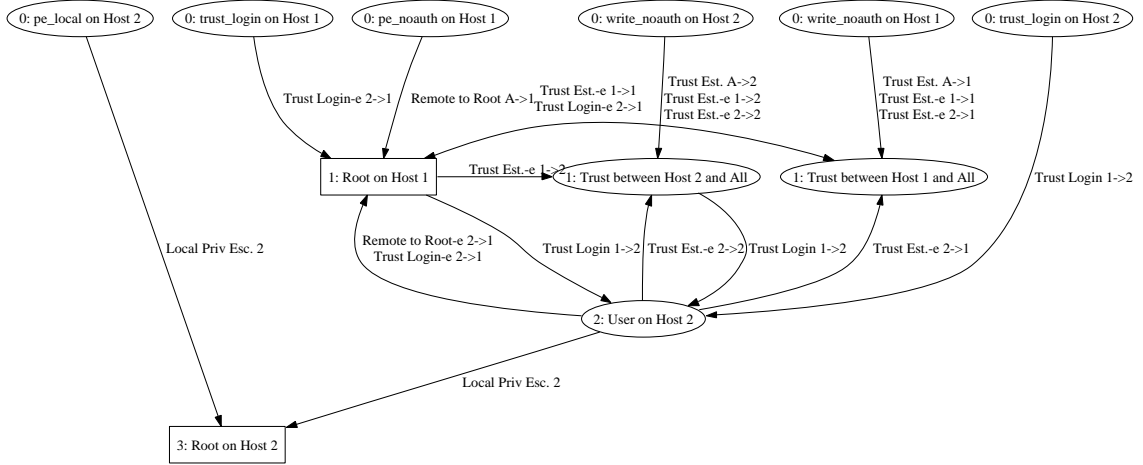


Figure 6.1: Abstract model of the 3-node network presented in [1, 2, 3] with the initial conditions enumerated.

having a mixture of removed and watched edges. The fitness function returns *benefit/cost* so that the more fit individuals are the ones which maximize the benefit and/or minimize the cost.

A genetic algorithm using deterministic rank selection and single point crossover has been selected to evolve the population. For purposes of the crossover operation, all three bitmaps are viewed as a single contiguous bit string. The crossover will occur at some random point along that string. For the mutation operation, a randomly chosen single bit along the string is flipped. This will result in the activation or removal of hardening measures. The initial population is generated by randomly setting the bits in the string.

6.4 Experiments

Consider the three node network presented in [1, 2, 3] and shown in Figure 6.1. Several previous works [26, 28, 23] use this network as an example. By the provisions of [1, 2, 3], RSH login (“Trust Login” in the abstract model), only occurs between Hosts 1 and 2 it would be detected by the IDS if the attacker tried it from his machine (Host A). Thus, the attack graph in Figure 6.1 only has Trust Login between Hosts 1 and 2. Were this provision removed, it would also be possible to do Trust Login between Host A and Host 1 once trust is established on Host 1 and likewise for Host 2.

Let us consider a scenario with this network. The administrator wishes to prevent an attacker from gaining root privileges on Host 2, but does not care if the attacker gains user privileges on Host 2 or root privileges on Host 1. Thus the goal nodes will be obtaining root on Host 2. The policy of the network is to allow RSH between the two hosts, allow SSH to Host 1 and allow FTP to both hosts. As a consequence of this policy, there will be no possible firewall rules for this network. There are patches available for the vulnerabilities “pe_noauth”, “write_noauth”, and “pe_local”. This means there are four possible initial conditions to patch: “pe_noauth on Host 1”, “write_noauth on Host 1”, “write_noauth on Host 2” and “pe_local on Host 2”. To make a comparison to the prior works, for this analysis IDS sensor placement will not be considered as the prior works did not consider such. Thus, this analysis will purely consider patching the four patchable initial conditions. This results in $2^4 = 16$ possible chromosomes. This is a rather trivial example as it is highly likely the initial population will have all possible chromosomes, thus the highest fitness in the initial population will likely be the optimal solution. However, this example was chosen so that it could be compared to the prior works.

It is possible to enumerate all chromosomes by hand for this example. With a larger chromosome, this would be infeasible. All possible chromosomes are listed in Table 6.1. There is only one goal node and that goal node has only one incoming attack edge (“Local Priv Esc”) so the maximum benefit is 1. The cost of patching any of the vulnerabilities is set at 1, so that the cost becomes the number of patches that would need to be installed. For more complex examples, the cost for each patch may differ so that two low cost patches might be cheaper than one high cost patch. The equal cost of patching is again a reflection of the assumptions in prior works and is set here so that a comparison to the prior works can be made. From the enumeration, it is clear there are three equally optimal patching solutions: “patch pe_local on Host 2”, “patch write_noauth on Host 2” and “patch pe_noauth on Host 1”. Figure 6.4 shows the effect of patching pe_local on Host 2. The dashed lines and nodes indicate all nodes and edges removed from the attack graph by this patch. Figure 6.4 shows the effect of patching write_noauth on Host 2 and Figure 6.4 shows the effect of patching pe_noauth on Host 1.

Translating the abstract model labels back to the exploit based model labels, these

Table 6.1: The enumeration of all possible patches for the 3 node network in Figure 6.1.

Chromosome	Patches	Cost	Benefit	Fitness
0000	None	0	0	0
0001	pe_local 2	1	1	1
0010	write_noauth 2	1	1	1
0011	write_noauth 2, pe_local 2	2	1	0.5
0100	write_noauth 1	1	0	0
0101	write_noauth 1, pe_local 2	2	1	0.5
0110	write_noauth 1, write_noauth 2	2	1	0.5
0111	write_noauth 1, write_noauth 2, pe_local 2	3	1	0.33
1000	pe_noauth 1	1	1	1
1001	pe_noauth 1, pe_local 2	2	1	0.5
1010	pe_noauth 1, write_noauth 2	2	1	0.5
1011	pe_noauth 1, write_noauth 2, pe_local 2	3	1	0.33
1100	pe_noauth 1, write_noauth 1	2	1	0.5
1101	pe_noauth 1, write_noauth 1, pe_local 2	3	1	0.33
1110	pe_noauth 1, write_noauth 1, write_noauth 2	3	1	0.33
1111	All	4	1	0.25

patches correspond to: “patch the xterm buffer overflow on Host 2” or “patch the FTP writable directory vulnerability on Host 2” or “patch the SSH buffer overflow vulnerability on Host 1”. The installation of any of these patches will prevent the attacker from obtaining root on Host 2 in the network model.

Now to compare these results to those in the prior works. The algorithm of Noel, et al. [26, 28] derived the strategies “prevent ftp(0,2), prevent ftp(0,1) and prevent ssh(0,1)” or “prevent ftp(0,2) and prevent ftp(1,2)” to prevent obtaining root on Host 2. Ammann’s [23] algorithm returns three paths:

1. ssh(0,1), ftp(1,2), rsh(1,2), xterm(2,2)
2. ftp(0,2), rsh(0,2), xterm(2,2)
3. ftp(0,1), rsh(0,1), ftp(1,2), rsh(1,2), xterm(2,2)

It should be noted that both these analyses ignore the provision in [1, 2, 3] that the RSH login from the attacker (machine 0 in prior works, Host A in this work) to Hosts 1 and 2 is detectable and thus not chosen. When this provision is enabled, the algorithm of Noel, et al. [26, 28] produces “prevent ftp(1,2)” or “prevent ssh(0,1)” and Ammann’s [23] algorithm would only return the first path.

Figure 1 is a complex dependency graph illustrating the relationships between various entities and their trust levels. The graph is divided into three main sections: Local Priv Esc. 2, Trust Login 1->2, and Trust Login 1->1. The graph shows a complex web of dependencies and trust levels between these entities.

Local Priv Esc. 2: This section includes nodes for local privilege escalation (0: pe_local on Host 2) and root access (1: Root on Host 2). The graph shows a complex web of dependencies and trust levels between these entities.

Trust Login 1->2: This section includes nodes for trust login (0: trust_login on Host 1, 0: trust_login on Host 2) and user access (2: User on Host 2). The graph shows a complex web of dependencies and trust levels between these entities.

Trust Login 1->1: This section includes nodes for privilege escalation (0: pe_noauth on Host 1, 0: write_noauth on Host 1) and root access (1: Root on Host 1). The graph shows a complex web of dependencies and trust levels between these entities.

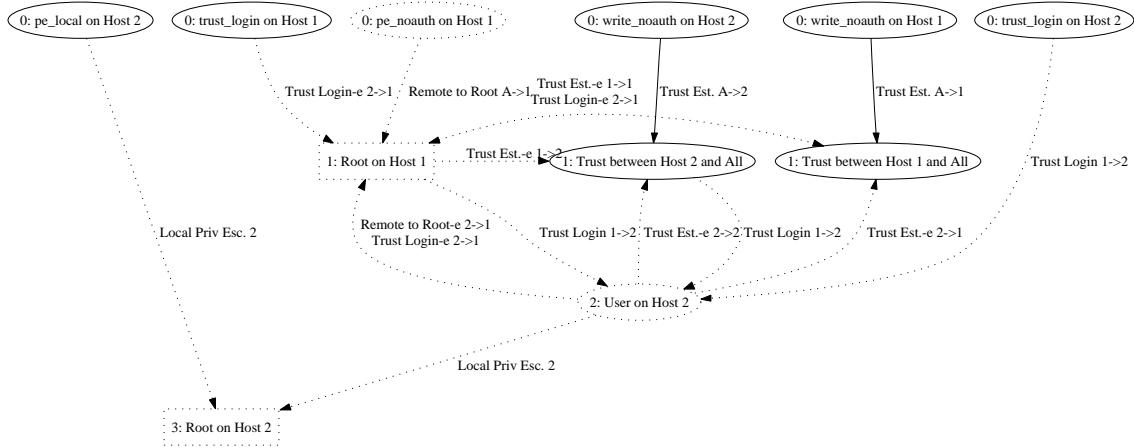


Figure 6.4: The effects of patching the `pe_noauth` vulnerability on Host 1. The patched node is dashed as are any nodes or edges disabled by this patch.

Noel, et al. [26, 28] did not show that one could also patch the `xterm` buffer overflow vulnerability to prevent the attacker from gaining root on Host 2. There is not an indication in their paper as to why this was not considered as a possible solution. Otherwise, their solution is identical to the genetic algorithm analysis. Ammann’s [23] algorithm does not provide explicit patching recommendations, but it does return all possible exploit paths. In this example, there is only one possible exploit path. With the policy that `rsh` could not be affected, the administrator is then left with the choice to prevent `ssh(0,1)`, `ftp(1,2)` or `xterm(2,2)` to remove that exploit path. This corresponds to the optimal solutions obtained by the genetic algorithm. Thus on this small network example, the genetic algorithm analysis obtains the same solutions as the previous works.

6.5 Conclusion

Previous analysis methods have ranged from the “try and test” approach using a simulation tool [24], greedy approximation algorithms [1, 2, 3, 23] and expressing the graph as a logical expression of the initial capabilities [26, 28]. The “try and test” method requires extensive interaction by the user. The greedy approximation algorithms of [1, 2, 3] have runtimes that are exponential in terms of the size of the network and their methods do not scale well to larger networks. Ammann’s [23] method is quadratic in terms of the number

of attacks and it only finds the set of minimal critical attacks, not the lowest cost set of hardening measures. The logical expression algorithm of [26, 28] only looks at patching the initial vulnerabilities, not at applying other hardening measures.

On the simple network example of Figure 6.1, this approach obtains the same results as the prior works of Noel, et al. and Ammann [26, 28, 23]. In the case of the prior works of Noel, et al. [26, 28], this approach found more solutions than their approach because their approach did not consider patching the local xterm buffer overflow vulnerability. Thus this approach does produce correct results when compared directly to other works, looking only at patching initial vulnerabilities in the network.

However, this approach is more flexible than the prior works. None of the prior methods consider the policy of the network when determining a set of hardening measures. Nor do they consider altering the firewall rules to eliminate network connectivity that is used for an attack but not needed by the policy of the network. The prior works also do not consider IDS placement to detect attacks that cannot be patched or firewalled. The genetic algorithm considers all of these factors when determining possible strategies to secure the network.

Chapter 7

Future Work

7.1 Introduction

The artificial immune system (AIS) was able to achieve a good false negative rate, but there is still room for improvement. The future work in AIS focuses on improving the false negative rate. Additionally, affinity maturation increases the specificity of an antibody, but this reduces the ability of an antibody to detect variants of an attack. Generalizing antibodies would allow them to detect a wider range of variants. This would also reduce the false negative rate.

Three areas of future work for attack graphs are complexity, visualization and analysis. Complexity of the Rete algorithm is greatly affected by the number of rules, so reducing complexity focuses on reducing the number of active rules during the attack graph computation. The primary issue with visualization is the visual complexity of attack graphs, even when the network consists of less than 50 hosts. The future work for visualization focuses on methods to further reduce visual complexity. For analysis, the future work expands the technique from Chapter 6 to analyze attack graphs for other purposes such as forensics analysis.

Section 7.2 details future work on artificial immune systems. Section 7.2.1 focuses on the idea of generalizing the artificial immune system antibodies to detect a wider range of attack variants. Section 7.3 explores three areas of future work for attack graphs: computation, visualization and analysis.

7.2 Artificial Immune Systems

While the false negative rate is good for the best tests for the AIS, it is still a bit high. The attack sample set was very small which allows an attack that is difficult to distinguish from normal requests to have a large effect on the false negative rate. Larger sample sets would likely yield better false negative rates by reducing the effect of such difficult to detect attacks. More attack instances are currently being gathered to see if a larger sample set does indeed reduce the false negative rate.

Improvement in the false negative and false positive rates may also be obtained by implementing a true multilevel immune system with separate sets of B cells and helper T cells which must collaborate together in order to detect attacks. The use of the tuning parameters somewhat approximates this but without separating the detectors into two different groups. Such a system would also be conducive to developing a peripheral tolerance by implementing a cell death if it doesn't get a collaboratory signal within a timeout period or to develop something akin to the cytotoxic T cells to destroy overly self-reactive cells.

Similarly, it may also be possible to decrease the false positive rate without significantly increasing the false negative rate by correlating between populations. On the best runs, most populations missed nearly the same attacks. However, the normal instances that were mislabeled as attacks seemed to vary more between populations. Requiring multiple populations to agree on an attack could remove many false positives without missing many more attacks.

It is also possible that a different matching function could improve this system. As shown in Gonzales *et al.* [45], the matching function can greatly affect the non-self coverage space. The use of a matching function which returns an affinity measure may yield better results, especially if this measure was also incorporated into the fitness of the antibodies during the breeding phase. Instead of using the number of attacks detected as the fitness function, a fitness function based on affinity such as the average affinity of the best affinity may be better.

One issue with the fitness function is that it only considers antibodies which directly react to attacks in the training dataset. This results in some loss of generalization

when the most fit members of the population are chosen as parent antibodies and survivors for the next generation. As a result, the antibodies may not be generalized enough to detect variants of the attacks in the data set. Variants occur quite often in malicious code. These variations can be the result of another person altering attack code that is already in circulation or an attack which mutates itself. Section 7.2.1 goes into details on methods to generalize the antibodies to cover more attack instances. The generalization phase could occur during the survivor selection portion of the lifecycle, taking a certain percentage of survivors and generalizing them for the next generation. It can also take place at the end of the lifecycle to generalize the final antibodies.

7.2.1 Generalizing Antibodies

The challenge is to generalize the detection ability of the antibodies so that they might detect variants without causing them to react to normal traffic. Whether this is possible depends on how closely the variants resemble each other once they have been abstracted into the fingerprint. As shown with Code Red in Section 2.6.4, some variants may abstract to the same fingerprint, but this is not often the case. As mentioned in Section 2.2.2, often man-made variants have several similarities because the person used one of the variants as a template. Polymorphic worms can also have some similarities in the code which handles the generation of variants. It is of course possible that the code would generate “terminal” variants which would not be able to further mutate themselves because they lack the code to generate variants. Such could be expected of a modular worm. These similarities may lead to similar fingerprints that encompass similar hyperspaces. The task then is to generalize the antibodies to cover a larger hyperspace so that they might detect the variants.

One method the biological immune system uses to detect antigens that are similar in structure is to have antibody receptors which are cross-reactive. These receptors may have low affinity to the various antigens, but this does offer a chance at detecting the antigens. The cross-reactivity threshold can be thought of as the radius of a hyperspace around the detector. Within this hyperspace, an antigen can be detected, as shown in Figure 7.1. In the artificial immune system, ideally the cross-reactivity threshold will be generalized enough

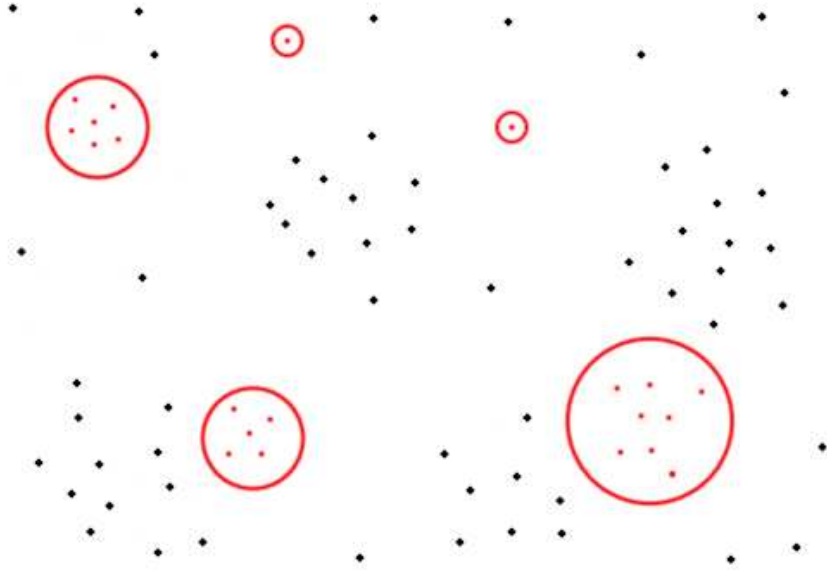


Figure 7.1: An example of the coverage space of cross-reactive antigens.

to detect the majority of variants. While [43] represents the cross-reactivity threshold as one value applied uniformly to all attributes, in this method each attribute has its own threshold.

Inductive learning algorithms from machine learning are one method that could be used to generalize existing antibodies without making them detect normal data. Inductive learning algorithms take a set of positive and a set of negative instances and derive a generalized description of the positive version space. For this application, the positive instances would be the attack data and the negative instances would be the normal data. The derived version space would be a generalized description of the attacks detected by an antibody that does not overlap into the normal data space.

However, an issue with most inductive learning algorithms is that they do not handle continuous attributes. A common workaround to this limitation is to discretize the continuous attributes. However, discretizing the continuous attributes in the antibodies would not dovetail well with the rest of the framework. Some evolutionary computation algorithms have been proposed to deal with continuous attributes in inductive learning algorithms.

Prior Work

Divina *et al.* [64] propose a method to handle continuous attributes that is based on statistical properties and an evolutionary algorithm. Continuous values are represented as a constraint on the allowed values for that attribute in the form $a \leq X \leq b$. To create the constraints, clustering using a Gaussian distribution based algorithm is performed on the positive examples which creates a set of clusters. The constraint boundaries, a and b , for any constraint are both members of a single cluster. Constraints can be created, deleted or modified during the evolutionary algorithm. The modification operators defined are enlarge, shrink, ground, shift and change cluster. The enlarge operator potentially widens the constraint boundaries such that the new boundaries a' and b' have the properties $a' \leq a$ and $b \leq b'$. The enlarge operator serves to generalize a constraint. The shrink operator potentially narrows the boundaries such that $a \leq a'$ and $b' \leq b$. The ground value makes the constraint a constant value by having $a' = b'$. The shrink and ground operators serve to specialize a constraint. The shift operator chooses new boundaries such that the probability that the attribute is between the new boundaries is equal to the probability that it was between the original boundaries. For all the preceding modification operators, the new boundaries are within the same cluster as the original boundaries. However, for the change cluster operation the new boundaries are chosen from a randomly selected different cluster. The new boundaries are chosen at random from this new cluster. The shift and change cluster operators serve to mutate the constraint. The implementation of this algorithm that was presented in the paper was embedded in ECL. During the evolutionary process of this algorithm, some of the above operators are applied to a constraint and the one which yields the best fitness is chosen as the change to the constraint [64].

Another method to handle continuous attributes was presented by Dasgupta and Gonzales [65]. Their system uses a set of rules to define the attack space. The rules consist of a condition which defines a hyperspace in the attack space. The condition consists of the conjunction of attribute boundaries. Similar to Divina *et al.*, the boundaries are high and low values for the attribute. Dasgupta and Gonzales also define a variability parameter that controls how close to normal space the attack space hyperspaces are allowed to be.

A genetic algorithm is used to derive the rules using different values for the variability parameter. The genetic algorithm is given only a set of normal instances and it outputs a set of rules that should cover the attack space. It generates the ruleset by a niching algorithm where the genetic algorithm is run multiple times. The basic fitness of a rule is the volume of its hyperspace. The fitness is reduced by the number of normal instances contained in its hyperspace and by the volume of the overlap between its hyperspace and the hyperspace of previously generated rules [65].

Cross-Reactivity Method

I also propose a method to generalize the attributes that does not require the statistical information of Davina *et al.*, since the number of variants in the attack dataset may be too small to derive meaningful statistics, and uses a variability parameter similar to Dasgupta and Gonzales. Also, since the sole purpose of this method is to generalize the antibodies and the antibodies should not react strongly to self due to the negative selection phase, only the generalization operation of expanding the boundaries need to be defined. In this method, the new boundaries for the generalization operands are chosen deterministically based off the variability parameter and randomly. This method uses the function *near* and the operand *expand* that are defined as follows.

The *nearness* function defines how close to an antibody an instance is. This function only checks the expressed attributes in the antibody. For discrete attributes, if the value of the attribute for the antibody does not match the value of the attribute for the instance, then the instance is not near the antibody and the function returns a “not near” value such as -1 or MAX_INT. For the continuous attributes, the function simply returns the Euclidean distance from the center of the antibody’s attribute ranges.

The *expand* operation generates several candidate boundary changes to the antibody. The candidates are compared to the original antibody and the one with the best fitness is selected. The purpose of this operation is to expand the expressed continuous attributes to cover variants without expanding into the normal space. One could consider the normal space when determining new boundaries, but that task is left to the fitness function and the negative selection phase.

The deterministic portion of *expand* takes the current range of the attribute and increases that range by the variability parameter, p . It generates three candidate boundary changes: $(a - p, b)$, $(a, b + p)$ and $(a - \text{floor}(\frac{p}{2}), b + \text{ceil}(\frac{p}{2}))$. Thus if an attribute's range is 4 to 5 and the variability parameter is 4, the deterministic method produces the following candidates: (0,5), (4,9) and (2,7).

The random portion of *expand* generates multiple candidate boundary changes. The first candidate change randomly selects a starting point $a - p < a' < a$ and adjusts b according so that the range is increased by p . The second boundary change similarly randomly selects an ending point from $b < b' < b + p$ and adjusts a accordingly. If either of these generated candidates is equivalent to the other or the deterministic $(a - \frac{p}{2}, b + \frac{p}{2})$ candidate, it is discarded. Using the prior example, this might generate the candidates (3,8) and (1,6).

The random method also randomly generates a secondary variability parameter, $0 < p' < p$. All five above listed formulas for calculating new candidates are repeated using p' . Using the prior example, let $p' = 3$. The deterministic formulas would create the candidates (1,5), (4,8) and (3,7) and the random formulas might produce the candidate (2,6), with the other candidate being discarded as a duplicate.

So, for the attribute example, the original range was (4,5) which is compared to the candidate ranges (0,5), (4,9), (2,7), (3,8), (1,6), (1,5), (4,8), (3,7) and (2,6). The range with the best fitness is chosen as the new range for the attribute.

Since evolutionary algorithms are used, another fitness function would be needed for the generalization phase. Several fitness functions are considered. One fitness function is the inverse of the number of normal instances covered by the antibody. Thus the fewer normal instances covered, the better the fitness. However, this doesn't take into account the coverage of attack instances which may be desirable if the data set does contain variants. Another fitness function takes the number of attack instances covered and subtracts the number of normal instances covered. An issue with this function is that it does not take into account the remainder of the version space, just specific instances in it. Two other fitness functions were proposed in the literature. Dasgupta and Gonzales propose a raw fitness function that is the volume of the attack space minus the number of normal instances

covered in the attack space [65]. Divina *et al.* propose the following fitness function [64]:

$$\frac{\text{attacks covered} + (\text{total normal} - \text{normal covered})}{\text{total examples}}$$

All of these fitness functions will be investigated to see which performs best. It is possible that different ones will be better for the different methods.

7.3 Attack Graphs

7.3.1 Computational Complexity

As noted in Section 5.2, the computational complexity of the Rete algorithm is greatly affected by the number of rules. By abstracting the atomic attacks into the abstract attack model, the number of rules was significantly reduced. A further reduction in the number of rules could be achieved by dynamically loading only the rules needed for the network being evaluated.

There are two types of rules in the model: those which depend on the postcondition(s) of other rules and those which only depend on the initial capabilities and connectivity in the network description. The former set of rules will always need to be loaded because it is not known at the start of the evaluation which postconditions that the rules depend on will be enabled. The later set of rules however can be filtered based on the capabilities in the network description. Any such rules which rely on a capability that is not in the network description will never be enabled and thus can be removed without affecting the results.

As shown in Figure 5.1, even reducing the model by as little as 2 rules greatly reduces the theoretical worse case runtime of the model significantly. By removing unnecessary rules from the Rete algorithm, the computational time will decrease.

7.3.2 Visualization

One issue with the usefulness of attack graphs to an administrator is the visualization of larger networks. Typically, there are so many nodes and edges in the attack graphs of all but the most trivial networks that the graph is difficult to read. Clustering identical machines as described in Chapter 5 helps reduce not just the computational complexity




Figure 7.2: The attack graph for a 56 host network that was clustered into 40 machine clusters.

but also the visual complexity by aggregating identical nodes and edges into one metanode. However, even with identical machine clustering, the graph quickly becomes cluttered. For example, Figure 7.2 shows a network that originally had 56 hosts that were clustered into 40 hosts. Even for this small network, the graph displays so much information that it is difficult to read. Further aggregation beyond identical machine clustering should reduce the visual complexity.

Prior Work

Noel and Jajodia [27] present a method to aggregate attack graphs through a hierarchy of rules based on the semantics of attack graphs. At the bottom of the hierarchy, exploits are aggregated into exploit sets if they have the same $\{\text{source}, \text{target}\}$ host pair and conditions are aggregated into condition sets if they are all either preconditions or postconditions of the same exploit. At the next level, condition sets are aggregated into machine sets if they are all conditions on the same host machine or condition sets are aggregated into an exploit set if all the conditions correspond to exploits in the exploit set. At the next level of the hierarchy, connected subgraphs of machine and exploit sets are aggregated into machine-exploit sets and machines are aggregated into protection domains if they are on the same network subnet. At the highest level, protection domains that form a connected subgraph are aggregated into a protection domain set. They showed how this aggregation hierarchy reduced an attack graph with hundreds of nodes to a protection domain set level graph of less than a dozen nodes.

Statecharts [66, 67, 68] are a method to aggregate states in a finite state machine into hierarchical superstates. Statecharts were designed for reactive systems that are event driven. The transitions correlate to the events that trigger a change in state. Clustering allows statecharts to combine related states into a superstate. While visualized as a superstate, only transitions which enter and exit the superstate are shown. Internal states and

transitions remain hidden until the superstate is refined into its constituent states. Statecharts also model orthogonality more concisely than with finite state machines, allowing one to concisely represent an AND condition with a single superstate. Appendix B has a more detailed description of statecharts and its formalisms.

Approach

The formalisms of statecharts provide several advantages to use with attack graph visualization. The work of Noel and Jajodia [27] is one way to add hierarchical aggregation to attack graph visualization. However, their highest aggregated mode has very little usable information beyond a count of the number of machines and exploits found in that protection domain set. One must deaggregate the graph down to the lowest levels to obtain information about the exploits executed. Statecharts however provide for conditional transition into a superstate that could be used to retain information about exploits entering a superstate while hiding the details. For example, the event would be the exploit label while the conditions would be {source,target} pair. Thus the aggregation could retain information about the exploits entering and leaving a superstate, while aggregating the details within the superstate. This method could be combined with the analysis to highlight the paths and nodes that the analysis has deemed crucial while aggregating the other paths and nodes into superstates.

7.3.3 Analysis

The analysis method presented in Chapter 6 focused primarily on analyzing the attack graph to determine a set of patches to apply, a set of firewall rules to add and a set of exploits that should be added to the intrusion detection system. Priority amongst these choices can be altered by altering the fitness weights applied to each. Attack graphs can be used for other purposes however, such as forensics analysis and response. Methods to analyze the attack graph for these purposes varies from the analysis presented in Chapter 6.

Forensics analysis matches the evidence gathered against an exploratory graph of the network. The analysis should produce likely goal nodes or further exploit paths given

the current evidence. This gives direction to the investigation, pointing out likely paths in the network that may have been taken. Response is similar in that it also matches evidence against an exploratory graph of the network. It also produces likely goal nodes and/or further exploit paths. With response however, the outcome of the analysis is protective or reactive measures to put in place to prevent the attacker from reaching the possible goal nodes or continuing down further exploit paths. Response has an additional layer of automated analysis to derive these protective measures while forensics just presents the likely goal nodes and exploit paths to the analyst for further investigation.

For both modes of analysis, the first stage is to derive the plausible goal nodes and exploit paths. An exploratory attack graph of the system could be generated at the time of analysis, but this will add to the time the analysis will take. For response, time is of an essence as any delay may afford the attacker time to progress further into the network. Since the exploratory attack graph of the network only changes when machines, network connectivity, firewalls and IDS placement are altered, the attack graph could be computed when one of these events occurs then stored for use in forensics and response analysis. To derive plausible goal nodes and exploit paths, evidence is matched to nodes and edges in the exploratory attack graph. Evidence can consist of noted states on machine(s) in the network which would correlate to nodes in the attack graph and of observed attacks which would correlate to edges in the attack graph. Paths in the attack graph which do not match any evidence can be pruned. The remaining graph after pruning is the plausible exploit paths. Any goal nodes in the remaining graph are the plausible goal nodes.

Response analysis would then take this reduced graph of plausible exploit paths and determine a set of response measures to take. This analysis would be similar to Chapter 6, using evolutionary computation to determine a low cost and high benefit set of response measures. The chromosome would need to be altered to cover the response measures rather than patching, firewalling and placing IDS sensors.

Chapter 8

Conclusion

This work focused on the topics of detecting attacks against computer systems and managing computer systems to prevent and react to attacks. For detecting attacks, an artificial immune system (AIS) was developed. Most current AIS algorithms [15, 16, 17, 18, 19, 20] focus solely on negative selection to create detectors. This leads to a high false negative rate for certain types of attacks that may closely resemble normal behavior. The concepts of positive selection and collaboration were added to reduce the false negative rate without unduly increasing the false positive rate. Positive selection allows detectors which have a slight reactivity to normal behavior, whereas pure negative selection would destroy such detectors. Collaborations requires that multiple detectors agree to label data an attack before it is labeled an attack. A higher positive selection value always led to more attacks being detected which is a lower false negative rate, but also led to more false positives. A higher collaboration value decreased the false positive rate, but increased the false negative rate. The two values worked together synergistically in that the collaboration value did not completely counteract the effects of the positive selection value. The collaboration value had a stronger effect on reducing false positives, than it did on increasing the false negatives. This is a desired result. By combining these two features, a higher number of attacks can be detected without greatly increasing the number of false alarms.

Attack graphs are one method for managing computer systems to prevent attacks. Attack graphs can facilitate the design of new networks to determine how to segment the

network and place IDS sensors to minimize risk. Attack graphs can also be used to analyze existing networks to determine a risk mitigation strategy. Additionally, attack graphs can be used to assist forensics on a network after an attack has been detected and to determine automatic response measures to an ongoing attack. This work focused on five problems with attack graphs: scalability to large networks, ease of coding new attacks into the model, dealing with incomplete network information, visualization of attack graphs and automatic analysis of attack graphs.

Scalability to modern sized networks is one of the biggest challenges within attack graphs. If the computation cannot be completed in a reasonable time, then attack graphs will not be useful to administrators. Prior works have focused on model checking [1, 2, 3, 22] or graph theory [23, 24, 25, 26, 27, 28] methods to automatically compute the attack graph. This work used the expert system JESS [5] which is based on the Rete [4] algorithm to compute the attack graph. Section 4.6.1 showed how the expert system approach was more scalable than the model checking approach on a small network. To gain further scalability, the abstract class model and clustering of identical nodes was implemented. By abstracting specific exploits into an abstract class, the number of rules for specifying attacks in the model are reduced. Clustering identical machines reduces the number of machines to evaluate by aggregating identical nodes into one metanode. The abstract model ran in less time than an equivalent exploit based model. Adding clustering to the abstract model yielded the greatest improvement in scalability. The expert system was able to analyze the test cases from Section 5.5.1 in about a minute. The largest amount of time was spent finding and aggregating the clusters in the preprocessing phase before passing the network description to the expert system. However, this preprocessing phase took significantly less time to run than the time it took the expert system to process the unclustered network information. With both the abstract model and clustering, networks containing thousands of hosts could be evaluated in a couple of hours on a standard Pentium 4 workstation.

The abstract model also makes coding new attacks into the model easy – important in reacting quickly to new attacks or vulnerabilities. If the attack falls under an existing abstract attack class, then the attack can be added to the model by adding its vulnerability to the list of vulnerabilities that fall under the abstract class. Otherwise, the

attack can be coded into the model using an expert system language that is supported by multiple expert systems. Incomplete network information is handled by supporting “what if” scenarios. The administrator can set certain vulnerabilities to explore the ramifications of those vulnerabilities being present in the system. Visualization has been decoupled from the automatic generation of the attack graph to allow the best methods for each to be chosen. Additionally, clustering identical machines helps reduce the visual complexity by greatly reducing the number of attack edges and capability nodes computed. For the larger networks evaluated, the number of edges with clustering was only about 2% of the number of edges without clustering. Even with this immense reduction in visual complexity, the resulting graphs can still be quite complex. Additional work in aggregating nodes and edges is proposed in Section 7.3.2.

The analysis methods presented in the prior works [1, 2, 3, 22, 23, 24, 25, 26, 27, 28] are less flexible than the analysis methods presented in this work. None of the prior works considers the policy of the system when computing the mitigation strategy. The burden of determining which of the proposed measures is appropriate for the system’s policy is placed on the administrator. For very large networks, there can be many different mitigation strategies derived by the methods in the prior work. To have to go through each by hand to decide which is appropriate for the policy is an undue burden placed on the administrator. This work incorporates the policy within the analysis, filtering out hardening measures that are not acceptable or that are undesirable with respects to the policy. The resulting mitigation strategy offers the most benefit for the least cost with respects to the system policy. Additionally, this work considers more hardening measures than the prior works. The prior works focused on patching vulnerabilities in the system while this work also adds the ability to add firewall rules or place IDS sensors to mitigate risk. The prior works had this focus because they only considered attack graphs being used for patch management. This work considers a wider use of attack graphs from patch management to intrusion response.

Appendix A

Artificial Immune System Framework

A.1 Program Description

The project was implemented in C++ on a Linux workstation using g++. The program was broken down into three main components: the antibody, the web data agent, and the lifecycle. There was one additional helper class for the lifecycle program; the antibody heap sort class. A makefile is provided which compiles all the components provided the system has g++ installed.

A.1.1 Antibody Class

This class implements all the basic functions of the antibody. It contains all the attributes and attribute expression flags. It has constructors that allow for the creation of a random antibody, an antibody according to given parameters or a clone of another antibody. The antibody class also tracks the fitness statistics for the antibody. The matching function, crossover routing and mutation function are also all members of the antibody class. The antibody class is not meant to be called directly. It should be compiled to a .o file and linked to a program which runs it.

A.1.2 Webdata Class

The webdata class contains all the training data for either the attack or normal traffic. An instance of the webdata class can only hold data for one of these types of traffic. The webdata class also has a test function which is called by the lifecycle during both the self-test and training phases. This function then selects entries from its data and submits that to the antibodies. It then checks if the antibody returned the proper response and updates the antibody's fitness statistics accordingly. The test function also accepts a flag which lets it know if the lifecycle is in training or self-test mode. This allows it to probabilistically select the normal traffic data as described in Section 2.6.1. It should also be compiled as a .o file and linked to the lifecycle program.

A.1.3 AntibodyHeap Class

The antibody heap class is simply a heap sort implemented to use the fitness function of the antibodies as the sorting function. It is a max heap sort, which facilitates the lifecycle's search for the antibodies with the best fitness in the breeding phase. It should also be compiled as a .o file and linked to the lifecycle program.

A.1.4 Lifecycle Program

The lifecycle program is what implements Figure 2.2. It takes two command line arguments: crossover rate and mutation rate. To change the number of antibodies per file requires the editing of a define and a quick "make". To run the lifecycle program, one needs to have a subdirectory off the lifecycle directory called "res" where the lifecycle program will store its results. Verbose output is undefined by default. To enable verbose output, define `VERBOSE_OUTPUT` and recompile. Warning: verbose output files are on the order of >100MB large for some input parameters. To run the lifecycle program with the crossover rate of 0.5 and the mutation rate of 0.1, type:

```
prompt$ ./lifecycle 0.7 0.1
```

at the command prompt and wait for it to finish. You can monitor the progress by looking at the contents of the files created in the "res" directory.

A.1.5 Output Files

There are 5 output files which are always created and the verbose output file which is only created when `VERBOSE_OUTPUT` is defined. The filenaming format is:

$$\text{output_type.}m_a\text{-}c_r\text{-}m_r$$

where m_a is the max antibodies, c_r is the crossover rate and m_r is the mutation rate.

The output types are as follows:

bad A count of the antibodies that failed the self-test phase

false_neg The false negative rates from the training phase

false_pos The false positive rates from the training phase

results The raw fitness statistics for each generation

summary The average fitness for each file and generation at the end of the training phase

A.2 Tester File

A small tester file is included which tests most features of the antibody without the computational overhead of running the lifecycle program. The tester file creates two random antibodies and outputs their values. It then breeds them to produce two children and outputs the children's values. It then runs a few mock tests on the matching routine. Finally, it performs a mutation. To run the tester, do the following:

```
prompt$ make test
prompt$ ./a.out
```

Appendix B

Statecharts

B.1 Introduction

Statecharts are a graphical, hierarchical formalism developed by David Harel to model reactive systems [66, 67, 68]. A reactive system is primarily event driven, reacting to both events in the outside world and internal events. A reactive system can be represented as a sequence of actions and events. States and transitions are a good method by which to model reactive systems, but traditional finite state machines are ill suited for such modeling. There is no hierarchical arrangements of the states; they are all “flat”. This leads to an exponential number of states because all states are visible at once. There is no way to represent a cluster of states as one superstate. Since there are no superstates, there is also a large number of transition arrows. A transition which may affect a group of states has to have an arrow to each individual state instead of just one arrow to the superstate. Concurrency is also not inherently expressable since state machines are sequential [66, 67].

B.2 Statechart Features

The main improvements that statecharts make over traditional state machines address the above shortcomings. Clustering allows statecharts to combine related states into a superstate. Refinement decomposes a superstate into individual states (or other superstates). Orthogonality allows for concurrency and independence. Broadcast communication

notifies all states of an event simultaneously [66].

The basic elements of a statechart are the states and transitions. States and transitions are labeled. The labels on the transitions correspond to events which cause the transition from the source state to the destination state. Events are described in more detail in Section B.2.4.

The following description of the statechart features comes from [66, 67].

B.2.1 Clustering and Refinement

When a set of states has some commonality, such as a shared output transition, they can be clustered (abstracted) into a superset. The superset represents the XOR of the set of states. Clustering allows for the number of states displayed to the user to be reduced by introducing a hierarchical method of displaying the states. One can look at the detailed view or a higher level abstraction. Clustering also reduces the number of transition arrows. This leads to a far more readable graph of the system. Clustering is a bottom-up procedure that starts with the most detailed representation and abstract this up into a higher level representation.

Refinement is the opposite process. In refinement, a superstate is decomposed into individual substates, which may themselves be superstates. This is a top-down procedure that starts with the most abstract representation and refines down to the detailed representation. Clustering and refinement gives statecharts a hierarchical form where states can be represented at many levels of detail.

B.2.2 Orthogonality

Orthogonality is the AND decomposition of states. It is represented by a state box with dashed lines between each element of the AND decomposition. Each element can be either a state or, typically, a superstate. When in an AND state, the system is in each one of the elements. Elements in an AND state can be synchronized when they share a set of transition events since all events apply to all substates in the AND state simultaneously. Elements may also behave independently by reacting to events to which the other elements do not react.

A big advantage introduced by the AND state is the more concise representation of the orthogonal product of the elements when compared to finite state machines. With finite state machines, each possible combination of elements at the most refined level must be represented. This leads to the large number of states mentioned in Section B.1. Thus, statecharts avoid the large number of states by using the more concise AND representation.

B.2.3 Entering States

When choosing which state to enter among a group of states, the state actually entered can be chosen by three primary methods. First, the transition may specify a state which is to be entered. Second, a default state may be indicated among the group of states. Third, the state may be chosen based on the history of prior entries. This group of states may be part of a superstate.

A default state is indicated by the default transition arrow notation. This notation is simply a transition arrow which does not originate from another state. Special care must be taken for the default arrow for a superstate. The notation must allow for easy refinement and clustering. For a superstate, the default arrow is a two stage process; one arrow pointing to the superstate and one arrow pointing to the default state within the superstate.

The simplest history entry chooses the most recently visited state of the group as the one to enter. The notation used to represent a history entry is a transition arrow that points to an “H” within the superstate. The history can apply on two levels. By default, it applies only to those states on the same level of the hierarchy as the history notation. The notation “H*” changes the default behavior so that the history applies to not only the states on the same level but also all substates of the current level. Making an analogy to trees, “H” applies to a specific level of the tree while “H*” applies to the leaves of the tree. History entries can be made more complex by basing the history on temporal logic or having a method by which the history can be cleared.

B.2.4 Events and Actions

Transitions between states are caused by events. Events can be either external events or events generated by the execution of the statechart. Transitions can be conditional

such that they are only taken when the event happens and the condition is true. The notation for conditional events is $\alpha(\beta)$, where α is the event and β is the condition. The condition can be a system condition such as checking if the system is in another state which can be used for synchronization between two AND states. The condition can also be a mathematical or boolean relation on system variables.

Transitions can also trigger internally created events, called actions. This is similar to Mealy machines. The notation for generating such actions is $\alpha(\beta)/\epsilon$, where α is the event, β is the condition (optional) and ϵ is the action. Actions occur instantaneously in zero time. Actions can also be triggered by the entry to and exit from states. Actions and external events are communicated to all states through a broadcast mechanism, so all states simultaneously react.

B.3 Syntax

This is a highlight of the most relevant portions of the syntax for statecharts. For the complex formal syntax, refer to [67].

A statechart is defined as a tuple (S, V, C, E, A, L, T) where S is the set of states, V is the set of expressions, C is the set of conditions, E is the set of events, A is the set of actions, L is the set of labels and T is the set of transitions.

Associated with states is the tuple (ρ, ψ, H, δ) where ρ is the hierarchy function, ψ is the type function, H is the set of history symbols and δ is the default function. The hierarchy function defines the substates for each superstate and has the form $\rho : S \rightarrow 2^S$. Two extensions of ρ are defined as $\rho^*(s) = \bigcup_{i \geq 0} \rho^i(s)$ and $\rho^+(s) = \bigcup_{i \geq 1} \rho^i(s)$. The type function says whether the state is an AND or OR composition and has the form $\psi : S \rightarrow \{AND, OR\}$. If a state s has $\psi(s) = OR$ and $\rho(s) \neq \emptyset$, it is an XOR superstate as described in Section B.2.1. If $\psi(s) = AND$ and $\rho(s) \neq \emptyset$, it is an AND superstate as described in Section B.2.2. The history symbols store the history for all type OR states which have a history entry. The function $\gamma : H \rightarrow S$ retrieves the state to enter for a particular history entry. The default function for a state s is the “set of states and history symbols which are contained in” [67] s and has the form $\delta : S \rightarrow 2^{S \cup H}$.

Associated with expressions is the set of variables V_p . Associated with the set of conditions is the set of primitive conditions C_p . Associated with the set of events is the set of primitive events E_p . See [67] for full definition of these sets.

Actions are defined as follows in [67], where 1-3 are atomic actions:

1. $\mu \in A$, μ is the null action.
2. If $c \in C_p, d \in C$ then $c := d \in A$ (assignment of conditions)
3. If $v \in V_p, u \in V$ then $v := u \in A$ (assignment of variables)
4. If $a_i \in A, i = 0, \dots, n$ then $a_0; \dots; a_n \in A$

The set of labels L are the labels on transitions. Labels are of the form $E \times A$ and a specified label is of the form $l = (e, a)$, with $l \in L, e \in E, a \in A$. This is written on the graph as e/a as described in Section B.2.4. The set of transitions has the form $T \subset 2^S \times L \times 2^{S \cup H}$. For a transition $t = (X, l, Y)$, X is the source set of states, Y is the target set of states or history entries and l is the label.

B.3.1 Definitions

The definitions are not covered in full detail. For full definitions, refer to [67].

Lowest Common Ancestor (LCA)

For a subset of S denoted S' , the state a which contains S' such that no substate of a contains S' . In formal terms, $LCA(X) = a$ iff $X \subseteq p^*(a)$ and $\forall s \in S \ X \subseteq p^*(s) \Rightarrow s \in p^*(a)$.

The strict lowest common OR ancestor is denoted $LCA^+(X)$. $LCA^+(X) = a$ iff $X \subseteq p^+(a)$, $\psi(a) = OR$ and $\forall s \in S$ if $\psi(s) = OR$ then $X \subseteq p^*(s) \Rightarrow s \in p^*(a)$.

The lowest common ancestor of a transition is the strict lowest common OR ancestor of the union of its source and target sets of states.

Orthogonal

Two states are orthogonal, denoted by \perp , if either they are the same state or their lowest common ancestor is an AND state. Formally, for $x, y \in S$, $x \perp y$ if $x = y$ or $\psi(LCA(\{x, y\})) = AND$.

An orthogonal set of states X contains only states which are orthogonal to each other. Formally, $\forall x_1, x_2 \in X \ x_1 \perp x_2$. An orthogonal set X is relative to a state s if $X \subset \rho^*(s)$.

A maximal orthogonal set X relative to state s is relative to s and $\forall y \in \rho^*(s) \ y \notin X \Rightarrow X \cup \{y\}$ is not orthogonal.

State Configuration

For a state s , its state configuration is an orthogonal set X that is relative to s such that $\forall x \in X \ \rho(x) = \emptyset$. The maximal state configuration of s is the set X that is a maximal orthogonal state relative to s such that $\forall x \in X \ \rho(x) = \emptyset$.

Structurally Consistent

Structurally consistent transitions are those which can be taken simultaneously. A structually consistent set of transitions T' contains only those transitions which are orthogonal to each other. Formally, $\forall t_1, t_2 \in T' \ LCA(t_1) \perp LCA(t_2)$.

Structurally Relevant

A set of transitions T' is structurally relevant to state configuration Z if $\forall t = (X, l, Y) \in T' \ \forall x \in X \ \exists z \in Z$ such that $z \in \rho^*(x)$.

B.4 Semantics

As with the syntax, not all details of the semantics presented in [67] are covered here. Only the most relevant portions to the execution of a statechart are presented.

The semantics are based on discrete time instances, denoted σ_i , which are organized into time intervals, denoted $I_i = [\sigma_i, \sigma_{i+1})$. When external stimuli occur in interval

I_i , the reaction takes place at time instance σ_{i+1} .

Definition: The external stimuli for time instance σ_{i+1} is defined as (π, θ, ξ) where $\pi \in E_p$ are the external primitive events that occurred during interval I_i , $\theta \in C_p$ is the external primitive conditions that are true and ξ gives the value of variables during the interval.

Definition: A system configuration for time instance σ_{i+1} is (X, π, θ, ξ) where (π, θ, ξ) is the external stimuli and X is a maximal state configuration for the root state. Refer to [67] for the definitions of how variables are evaluated, conditions are satisfied and events occur in the context of a system configuration.

Definition: A system reaction at some time instance σ_{i+1} is (Υ, Π) where Υ is a step and Π is the set of atomic events generated during the step.

A step is a chain reaction of transitions that occur at a time instance in response to external stimuli. All transitions occur simultaneously. No external stimuli can interrupt the chain reaction. To ensure this, all evaluation of variables and conditions are based on the values at the beginning of the reaction and all updates are done at the end of the chain reaction. Temporary chain reaction functions, called *current()*, are used to store the value changes to variables and conditions until the end of the reaction. The transitions in the step are consistent transitions that are structurally relevant to the system configuration at the time instance. The step is decomposed into a sequence of micro-steps.

Definition: A micro system configuration $\mu SC = (\mu X, \mu \pi, \mu \theta, \mu \xi, \mu Y)$ is related to a system configuration SC as follows:

1. $\mu X \subseteq X$, μX is a partial state configuration.
2. $\pi \subseteq \mu \pi$.
3. $\mu \theta \subseteq \{current(c) | c \in C_p\}$.
4. $\mu \xi$ assigns values to *current()* for variables.
5. μY is a partial state configuration.

Refer to [67] for the definitions of how variables are evaluated, conditions are satisfied and events occur in the context of a micro system configuration.

The possible successor, μSC , of a micro system configuration, μSC_1 , is defined as $\mu X \subseteq \mu X_1$, $\mu \pi_1 \subseteq \mu \pi$, and $\mu Y_1 \subseteq \mu Y$.

Definition: Inconsistent atomic actions are those which assign different values to the same condition or variable in a micro system configuration μSC . If two actions are not inconsistent with respect to μSC , then they are consistent. A non-atomic action $a_o; \dots; a_n$ is consistent if $\forall i, j \ 0 \leq i, j \leq n$ such that $i \neq j$, a_i is consistent with a_j .

A set of transitions $\mu \Upsilon$ which causes the action $a_o; \dots; a_n$ is consistent under μSC if $\mu \Upsilon$ is structurally consistent and the action is consistent.

Definition: A micro-step is a set of transitions $\mu \Upsilon$ that is consistent under μSC where for all transitions in the set, the transition is structurally relevant to μSC and the event which causes the transition occurs at μSC .

The first micro-step in a step is composed of those transitions which are triggered by SC . Taking the transitions in a micro-step results in a new micro system configuration, $\mu SC'$. This new configuration contains the results of executing the transitions in $\mu \Upsilon$ which includes new values to conditions, variables and expressions as recorded in the *current()* function and actions generated by taking the transitions. The transformation from μSC to $\mu SC'$ is only valid if any new events were generated by $\mu \Upsilon$, $\mu \Upsilon$ does not change a true condition in μSC to a false condition and $\mu \Upsilon$ does not change the value of any variable in μSC . Another micro-step is taken as long as there is a set of transitions that satisfies the definition of a micro-step under $\mu SC'$. If more than one such set exists, a set is selected non-deterministically. When there are no such sets of transitions remaining, then the step is over.

Therefore, a step Υ under system configuration SC is formally a maximal sequence of micro-steps $(\mu \Upsilon_0, \dots, \mu \Upsilon_m)$ where:

1. $\mu SC_0 = SC$.
2. $\mu \Upsilon_i$ is a micro-step from μSC_i for $i = 0, \dots, m$.
3. μSC_{i+1} is the micro system configuration created by executing $\mu \Upsilon_i$ for $i = 0, \dots, m$.
4. The set $\mu \Upsilon_0 \cup \dots \cup \mu \Upsilon_m$ is structurally consistent

5. If $\exists t \in T$ such that t is structurally relevant to $\mu\Upsilon_{m+1}$ and t 's event e occurs in $\mu\Upsilon_{m+1}$, then $\{t\} \cup \mu\Upsilon_0 \cup \dots \cup \mu\Upsilon_m$ is not structurally consistent.

The execution of the step causes the system configuration to update to reflect the changes that occurred during the micro-step and may also cause some events. These generated events are defined as $\Pi = \{e | e \text{ is generated by } \mu\Upsilon_i \text{ for } i = 0, \dots, m\} \cup \{e | e \text{ occurs in } \mu SC_{m+1}, e \text{ is atomic}\}$.

The entire run of the system is a sequence $\{(SC_i, \Upsilon_i, \Pi_i)\}$, where SC_i is the system configuration at point i , Υ_i is the step and Π_i is the set of events generated by Υ_i . The system is non-deterministic if for two runs, the sequence of steps differs or the sequence of generated events differs.

B.5 Additional Features

There are additional features to statecharts which were proposed by Harel [66] but not covered in the formal semantics [67]. Some of these features are used in the implementations of statecharts discussed in Section B.6.

B.5.1 Other Entrances

Conditional entrances into states can be used to simplify many conditional transitions which are based on the same action. For example, if all the substates in a superstate have conditional transitions based on the event α , rather than having a conditional transition for each substate, a conditional entrance can be used. The notation for a conditional entrance is a "C". A transition labeled with the event points to the "C". Transition arrows labeled with the condition go from "C" to the specific states reached by $\alpha(\text{condition})$. Selection entrances allow some value associated with an event to select which state of a group of states to enter. The notation for selection entrances is "S". It is represented similarly to the conditional entrance.

B.5.2 Timeouts and Delays

A timeout transition is a special event which occurs a specified number of time units after a triggering event. The notation for a timeout assumes the triggering event is entering the current state. Delays ensure that no transitions are taken to leave the delayed state until after the delay is passed. Delays can be thought of as lower bounds on the time spent in a state while timeouts are upper bounds. The notation for timeouts and delays are combined. The notation is of the form $\Delta t_1 < \Delta t_2$ where Δt_1 is the delay and Δt_2 is the timeout. Either Δt_1 or Δt_2 can be omitted, indicating that the state has only a timeout or delay respectively. If the state has a timeout, then there should be a transition leaving the state to indicate what transition should be taken when the timeout event occurs.

B.5.3 Overlapping States

The statecharts formalism is very tree-like with no superstates sharing substates. However, there are many real world situations where substates are shared between two different superstates. This causes an OR relation between the two superstates instead of an XOR relationship. It is also possible that there are situations where a state (or superstate) may be in an AND relation with another state at some points during execution but not at other points. Rather than modeling the state twice, once alone and once in the AND relation, it is possible to use overlapping to model the state only once.

B.6 Implementations

STATEMATE and Unified Modeling Language(UML) are two implementations of statecharts. They combine statecharts with other methods of modeling to create a utility which can be used to model complex systems.

B.6.1 STATEMATE

STATEMATE method is intended to be used in system development, primarily in the specification phase. It combines statecharts with activity charts and module charts to model the behavioral, functional and structural aspects of the system respectively. The basic

use of statecharts by STATEMATE is as described here with the addition of the timeout extension discussed in Section B.5.2, the conditional and selection entrances discussed in Section B.5.1, scheduling actions for a future time and iterative actions. There are also extensions to tie the statecharts in with the activity charts [68].

B.6.2 Unified Modeling Language (UML)

UML uses a variation of statecharts called state diagrams along with other modeling techniques. State diagrams are object state machines. There are some major differences between the statecharts presented here and the state diagrams used in UML. UML state diagrams are intended to model the behavior of a type, instead of a process as with statecharts. Some of the main differences for UML state diagrams that this implies are:

- Parameterized events.
- Set directed communication instead of broadcast communication.
- No zero time assumption. Actions can take time.
- Since actions can take time, micro-step execution replaced with threads of execution.

UML state diagrams use the timeout extension described in Section B.5.2. UML state diagrams also employ final states to signal the end of execution for a superstate [69].

B.7 Related Work

Petri nets [70] are another method, originally proposed by Carl Petri, by which reactive systems can be modeled. Petri nets are usually expressed in a graphical format where the graph is composed of two types of nodes: transitions and places. Transitions are analogous to events. Places are analogous to conditions for events. Places can be marked with tokens to indicate that the condition is satisfied. The placement of tokens for the whole net is called the marking. Events can have input and output places. All input places must be marked in order for the event to be enabled. Events can only occur when they are enabled and they occur instantaneously. When an event occurs, a token is removed from

each input place and a token is put in each output place. This reflects the change of state that can happen when an event occurs. The execution of a petri net is represented by the sequence of events that occur and, consequently, the sequence of markings.

There is a hierarchical component to a petri net where a node may be refined down to a subnet or a collection of nodes may be abstracted into a node. Harel [66] remarks that this hierarchical decomposition is not satisfactory in that it does not lend itself to the high-level events and multiple levels of concurrency which statecharts can express. Since events in petri nets are nodes in the graph, an event cannot occur in multiple places in the graph in the same sense that it can in a statechart.

Zave [71] provides an alias extension to sequence diagrams which results in a system with an expressive power equal to statecharts [66]. A sequence diagram is a tree structure whose leaves are the inputs and whose nodes are the sequences. A node can be labeled with a human-readable description of the sequence. The children of a node can have multiple relations. The relation can be sequential where parent node is the result of the first child followed by the second child and so on. It can also be alternative where only one of the children leads to the parent node (XOR). It can also be iterative where the parent node is the result of zero or more occurrences of the child node. The leaf node takes a named input which can also be given an alias which is an interpretation of that input.

A sequence diagram is part of a view and an entire system is modeled by a set of views. This module also has a filter which handles input parsing and input error. An input may have multiple aliases. When the filter receives the input, it determines which aliases apply. It then sends the alias instead of the original input name to all views which use that alias. All views act upon the alias simultaneously, similar to how all relevant transitions in statecharts for an event occur simultaneously.

Appendix C

Exploit Based Attack Graph Rules

C.1 Rules from Published Papers

C.1.1 CMU Model

These rules are modified from the rules published in [1, 3]. The modifications make the rules monotonic and add the edge rules.

sshd-buffer-overflow: A buffer overflow attack on the SSHd daemon that gives root privileges to an attack with privileges less than root.

Main Preconditions

$var(t, ssh)$
 $priv(s) \geq user$
 $priv(t) < root$
 $connect(s, t, ssh)$

Postconditions

$priv(t) = root$

Edge Preconditions

$var(t, ssh)$
 $priv(t) = root$
 $priv(s) \geq user$
 $s \neq t$
 $\neg edge(s, t, ssh)$
 $connect(s, t, ssh)$

ftp-rhosts: A vulnerability in FTP allows anonymous clients to write to the home directory, allowing an attacker with no privileges to alter the `.rhosts` file.

Main Preconditions

$var(t, ftp)$
 $var(t, wdir)$
 $var(t, fshell)$
 $priv(s) \geq user$
 $\neg var(t, rshtrust)$
 $connect(s, t, ftp)$

Edge Preconditions

$var(t, rshtrust)$
 $var(t, ftp)$
 $var(t, wdir)$
 $var(t, fshell)$
 $priv(s) \geq user$
 $s \neq t$
 $\neg edge(s, t, ftp)$
 $connect(s, t, ftp)$

Postconditions

$var(t, rshtrust)$

remote-login: The RSH login attack exploits the trust relationship established with the `ftp-rhosts` attack to log into the machine with user privileges.

Main Preconditions

$var(t, rshtrust)$
 $priv(s) \geq user$
 $priv(t) = none$
 $connect(s, t, rsh)$

Edge Preconditions

$var(t, rshtrust)$
 $priv(s) \geq user$
 $s \neq t$
 $priv(t) > none$
 $\neg edge(s, t, login)$
 $connect(s, t, rsh)$

Postconditions

$priv(t) = user$

xterm-overflow: A vulnerability in the xterm local binary allows a user to elevate privileges to root.

Main Preconditions

$var(t, xterm)$
 $priv(t) = user$

Edge Preconditions

$var(t, xterm)$
 $priv(t) = root$
 $\neg edge(t, t, xterm)$

Postconditions

$priv(t) = root$

C.1.2 Sheyner Model

These rules are modified from the rules published in Sheyner's thesis [2]. The modifications make the rules monotonic and add the edge rules.

iis-buffer-overflow: A buffer overflow in the IIS web server allows an attacker to gain root privileges on the server when the attacker does not have root privileges.

Main Preconditions

$var(t, iis)$
 $priv(s) \geq user$
 $priv(t) < root$
 $connect(s, t, web)$

Postconditions

$priv(t) = root$

Edge Preconditions

$var(t, iis)$
 $priv(t) = root$
 $priv(s) \geq user$
 $s \neq t$
 $\neg edge(s, t, iis)$
 $connect(s, t, web)$

squid-port-scan: The squid proxy server allows an attacker to scan the network, regardless of the privileges the attacker has on the squid server.

Main Preconditions

$\neg var(scan)$
 $var(t, squid)$
 $priv(s) = user$
 $\neg edge(s, t, squid)$
 $connect(s, t, web)$

Postconditions

$var(scan)$

Edge Preconditions

$var(scan)$
 $var(t, squid)$
 $priv(s) = user$
 $\neg edge(s, t, squid)$
 $connect(s, t, web)$

licq-user-shell: An attacker can gain a user privilege shell via an exploit in the LICQ binary. The attack requires a port scan to have been done on the network.

Main Preconditions

$var(scan)$
 $var(t, licq)$
 $priv(s) \geq user$
 $priv(t) = none$
 $connect(s, t, licq)$

Postconditions

$priv(t) = user$

Edge Preconditions

$var(scan)$
 $var(t, licq)$
 $priv(s) \geq user$
 $s \neq t$
 $priv(t) > user$
 $\neg edge(s, t, licq)connect(s, t, licq)$

client-browser-trojan: A vulnerability in the client's web browser software allows an attacker to gain user privileges if the client visits the attacker's website.

Main Preconditions

$var(t, browser)$
 $priv(s) \geq user$
 $priv(t) = none$
 $connect(t, s, web)$

Postconditions

$priv(t) = user$

Edge Preconditions

$var(t, browser)$
 $priv(t) > none$
 $priv(s) \geq user$
 $s \neq t$
 $\neg edge(s, t, browser)$
 $connect(t, s, web)$

at-overflow: A vulnerability in the AT binary can allow a user to elevate their privileges to root.

Main Preconditions

$var(t, at)$
 $priv(t) = user$

Postconditions

$priv(t) = root$

Edge Preconditions

$var(t, at)$
 $priv(t) = root$
 $\neg edge(s, t, at)$

Appendix D

Abstract Attack Graph Rules

r2r-noauth: Remotely accessible service with a vulnerability that allows an attacker without any privileges to obtain root privileges.

Main Preconditions

$var(t, pe_noauth)$
 $priv(s) \geq user$
 $priv(t) < root$
 $connect(s, t, pe_noauth)$

Postconditions

$priv(t) = root$

Exploit Based Examples

sshd-buffer-overflow, iis-buffer-overflow

Edge Preconditions

$var(t, pe_noauth)$
 $priv(s) \geq user$
 $priv(t) = root$
 $\neg edge(s, t, r2r-noauth)$
 $connect(s, t, pe_noauth)$

r2r-auth: Remote accessible service with a vulnerability that allows an attacker with user privileges to obtain root privileges.

Main Preconditions

$var(t, pe_auth)$
 $priv(s) \geq user$
 $priv(t) = user$
 $connect(s, t, pe_auth)$

Postconditions

$priv(t) = root$

Exploit Based Examples

Edge Preconditions

$var(t, pe_auth)$
 $priv(s) \geq user$
 $priv(t) = root$
 $\neg edge(s, t, r2r-auth)$
 $connect(s, t, pe_auth)$

r2u: Remotely accessible service with a vulnerability that allows an attacker to obtain user privileges.

Main Preconditions

$var(t, pe_user)$
 $priv(s) \geq user$
 $priv(t) = none$
 $connect(s, t, pe_user)$

Postconditions

$priv(t) = user$

Exploit Based Examples

Edge Preconditions

$var(t, pe_user)$
 $priv(s) \geq user$
 $s \neq t$
 $priv(t) = user$
 $\neg edge(s, t, r2u)$
 $connect(s, t, pe_user)$

priv-esc: Local binary with a vulnerability that allows an attacker to escalate user privilege to root privilege.

Main Preconditions

$var(t, pe_local)$
 $priv(t) = user$

Postconditions

$priv(t) = root$

Exploit Based Examples

xterm-overflow, at-overflow

Edge Preconditions

$var(t, pe_local)$
 $priv(t) = root$
 $\neg edge(t, t, priv-esc)$

trust-est: Remotely accessible service with a vulnerability that allows an attacker to write to files used to authenticate users for trust based logins, *e.g.* the `.rhosts` file.

Main Preconditions

$var(t, write_noauth)$
 $priv(s) \geq user$
 $priv(t) = none$
 $\neg var(t, trust)$
 $connect(s, t, write_noauth)$

Postconditions

$var(t, trust)$

Exploit Based Examples

ftp-rhosts

Edge Preconditions

$var(t, trust)$
 $var(t, write_noauth)$
 $priv(s) \geq user$
 $priv(t) = none$
 $s \neq t$
 $\neg edge(s, t, trust-est)$
 $connect(s, t, write_noauth)$

trust-exp: Remotely accessible login service that uses trust for authentication and allows only user privilege logins.

Main Preconditions

$var(t, trust)$
 $var(t, trust_login)$
 $priv(s) \geq user$
 $priv(t) = none$
 $connect(s, t, trust_login)$

Postconditions

$priv(t) = user$

Exploit Based Examples

remote-login

Edge Preconditions

$var(t, trust)$
 $var(t, trust_login)$
 $priv(s) \geq user$
 $s \neq t$
 $priv(t) = user$
 $\neg edge(s, t, trust_exp)$
 $connect(s, t, trust_login)$

info-system-noauth: Remotely accessible service with a vulnerability that leaks system information to an attacker with no privileges.

Main Preconditions

$var(t, leak_noauth)$
 $\neg var(t, info_system)$
 $priv(s) \geq user$
 $priv(t) < root$
 $connect(s, t, leak_noauth)$

Postconditions

$var(t, info_system)$

Exploit Based Examples

Edge Preconditions

$var(t, leak_noauth)$
 $var(t, info_system)$
 $priv(s) \geq user$
 $priv(t) < root$
 $\neg edge(s, t, info_system_noauth)$
 $connect(s, t, leak_noauth)$

info-system-auth: Remotely accessible service with a vulnerability that leaks system information to an attacker with user privilege.

Main Preconditions

$var(t, leak_auth)$
 $\neg var(t, info_system)$
 $priv(s) \geq user$
 $priv(t) = user$
 $connect(s, t, leak_auth)$

Edge Preconditions

$var(t, leak_noauth)$
 $var(t, info_system)$
 $priv(s) \geq user$
 $priv(t) = user$
 $\neg edge(s, t, info_system-auth)$
 $connect(s, t, leak_auth)$

Postconditions

$var(t, info_system)$

Exploit Based Examples

info-system-local: Local binary with a vulnerability that leaks system information to an attacker with user privilege.

Main Preconditions

$var(t, leak_local)$
 $\neg var(t, info_system)$
 $priv(t) = user$

Edge Preconditions

$var(t, info_system)$
 $var(t, leak_local)$
 $priv(t) = user$
 $\neg edge(t, t, info_system-local)$

Postconditions

$var(t, info_system)$

Exploit Based Examples

r2r-info-system: Remotely accessible login service that allows root logins. To succeed, the attacker has to have obtained the root password via a system information leak and the root password must be recoverable from the system information, *e.g.* using a dictionary attack to recover the password.

Main Preconditions

$var(t, info_system)$
 $var(t, crackable_root_password)$
 $var(t, login)$
 $var(t, root_login)$
 $priv(t) < root$
 $priv(s) \geq user$
 $connect(s, t, login)$

Edge Preconditions

$var(t, info_system)$
 $var(t, crackable_root_password)$
 $var(t, login)$
 $var(t, root_login)$
 $priv(t) = root$
 $priv(s) \geq user$
 $\neg edge(s, t, r2r_info_system)$
 $connect(s, t, login)$

Postconditions

$priv(t) = root$

Exploit Based Examples

r2r-info-system-local: Local login binary that allows an attacker to change logins from user to root with a root password obtained from a system information leak as described in **r2r-info-system**.

Main Preconditions

$var(t, info_system)$
 $var(t, local_login)$
 $var(t, crackable_root_password)$
 $priv(t) = user$

Edge Preconditions

$var(t, info_system)$
 $var(t, local_login)$
 $var(t, crackable_root_password)$
 $priv(t) = root$
 $\neg edge(t, t, r2r_info_system_local)$

Postconditions

$priv(t) = root$

Exploit Based Examples

r2u-info-system: An attacker can log into remote login with a user password obtained from a system information leak.

Main Preconditions

var(t, info-system)
var(t, crackable_user_password)
var(t, login)
 $\text{priv}(s) \geq \text{user}$
 $\text{priv}(t) = \text{none}$
 $\text{connect}(s, t, \text{login})$

Edge Preconditions

var(t, info-system)
var(t, crackable_user_password)
var(t, login)
 $\text{priv}(s) \geq \text{user}$
 $s \neq t$
 $\text{priv}(t) = \text{user}$
 $\neg \text{edge}(s, t, \text{r2u-info-system})$
 $\text{connect}(s, t, \text{login})$

Postconditions

$\text{priv}(t) = \text{user}$

Exploit Based Examples

r2r-weak: Remotely accessible login service where root login is allowed and the system has a default, unset or brute force guessable root password.

Main Preconditions

var(t, login)
var(t, root_login)
var(t, weak_root_password)
 $\text{priv}(s) \geq \text{user}$
 $\text{priv}(t) < \text{root}$
 $\text{connect}(s, t, \text{login})$

Edge Preconditions

var(t, login)
var(t, root_login)
var(t, weak_root_password)
 $\text{priv}(s) \geq \text{user}$
 $\text{priv}(t) = \text{root}$
 $\neg \text{edge}(s, t, \text{r2r-weak})$
 $\text{connect}(s, t, \text{login})$

Postconditions

$\text{priv}(t) = \text{root}$

Exploit Based Examples

r2r-weak-local: Local login binary where root login is allowed and the system has a default, unset or brute force guessable root password.

Main Preconditions

$var(t, local_login)$
 $var(t, weak_root_password)$
 $priv(t) = user$

Edge Preconditions

$var(t, local_login)$
 $var(t, weak_root_password)$
 $priv(t) = root$
 $\neg edge(t, t, r2r-weak-local)$

Postconditions

$priv(t) = root$

Exploit Based Examples

r2u-weak: Remotely accessible login service that gives user privilege where the user has a default, unset or brute force guessable password.

Main Preconditions

$var(t, login)$
 $var(t, weak_user_password)$
 $priv(s) \geq user$
 $priv(t) = none$
 $connect(s, t, login)$

Edge Preconditions

$var(t, login)$
 $var(t, weak_user_password)$
 $priv(s) \geq user$
 $s \neq t$
 $priv(t) \geq user$
 $\neg edge(s, t, r2u-weak)$
 $connect(s, t, login)$

Postconditions

$priv(t) = user$

Exploit Based Examples

info-network-noauth: Remotely accessible service that leaks network information such as port scans to an attacker with no privileges.

Main Preconditions

$\neg var(info_network)$
 $var(t, leak_network_noauth)$
 $priv(s) \geq user$
 $priv(t) < root$
 $connect(s, t, leak_network_noauth)$

Edge Preconditions

$var(info_network)$
 $var(t, leak_network_noauth)$
 $priv(s) \geq user$
 $priv(t) < root$
 $\neg edge(s, t, info_network_noauth)$
 $connect(s, t, leak_network_noauth)$

Postconditions

$var(info_network)$

Exploit Based Examples

squid-port-scan

info-network-auth: Remotely accessible service that leaks network information to an attacker with user privilege.

Main Preconditions

$\neg var(info_network)$
 $var(t, leak_network_auth)$
 $priv(s) \geq user$
 $priv(t) = user$
 $connect(s, t, leak_network_auth)$

Edge Preconditions

$var(info_network)$
 $var(t, leak_network_auth)$
 $priv(s) \geq user$
 $priv(t) = user$
 $\neg edge(s, t, info_network_auth)$
 $connect(s, t, leak_network_auth)$

Postconditions

$var(info_network)$

Exploit Based Examples

info-network-local: Local binary that leaks network information to an attacker with user privilege.

Main Preconditions

$\neg var(info_network)$
 $var(t, leak_network_local)$
 $priv(t) = user$

Edge Preconditions

$var(info_network)$
 $var(t, leak_network_local)$
 $priv(t) = user$
 $\neg edge(t, t, info_network_local)$

Postconditions

$var(info_network)$

Exploit Based Examples

r2r-noauth-info: Remotely accessible service that gives root privileges to an attacker with no privileges using network information.

Main Preconditions

$var(info_network)$
 $var(t, pe_noauth_info)$
 $priv(s) \geq user$
 $priv(t) < root$
 $connect(s, t, pe_noauth_info)$

Edge Preconditions

$var(info_network)$
 $var(t, pe_noauth_info)$
 $priv(s) \geq user$
 $priv(t) = root$
 $\neg edge(s, t, r2r-noauth-info)$
 $connect(s, t, pe_noauth_info)$

Postconditions

$priv(t) = root$

Exploit Based Examples

r2r-auth-info: Remotely accessible service that gives root privileges to an attacker with user privilege using network information.

Main Preconditions

$var(info_network)$
 $var(t, pe_auth_info)$
 $priv(s) \geq user$
 $priv(t) = user$
 $connect(s, t, pe_auth_info)$

Edge Preconditions

$var(info_network)$
 $var(t, pe_auth_info)$
 $priv(s) \geq user$
 $priv(t) = root$
 $\neg edge(s, t, r2r-auth-info)$
 $connect(s, t, pe_auth_info)$

Postconditions

$priv(t) = root$

Exploit Based Examples

r2u-info-network: Remotely accessible service that gives user privileges to an attacker with no privileges using network information.

Main Preconditions

$var(info_network)$
 $var(t, pe_user_info)$
 $priv(s) \geq user$
 $priv(t) = none$
 $connect(s, t, pe_user_info)$

Edge Preconditions

$var(info_network)$
 $var(t, pe_user_info)$
 $priv(s) \geq user$
 $s \neq t$
 $priv(t) = user$
 $\neg edge(s, t, r2u_info_network)$
 $connect(s, t, pe_user_info)$

Postconditions

$priv(t) = user$

Exploit Based Examples

licq-user-shell

client-r2r-noauth: Client based attack that gives root privileges to an attacker with no privileges.

Main Preconditions

$var(t, client_pe_noauth)$
 $priv(s) \geq user$
 $priv(t) < root$
 $connect(t, s, client_pe_noauth)$

Edge Preconditions

$var(t, client_pe_noauth)$
 $priv(s) \geq user$
 $priv(t) = root$
 $\neg edge(s, t, client_r2r_noauth)$
 $connect(t, s, client_pe_noauth)$

Postconditions

$priv(t) = root$

Exploit Based Examples

client-r2r-auth: Client based attack that gives root privileges to an attacker with user privilege.

Main Preconditions

$var(t, client_pe_auth)$
 $priv(s) \geq user$
 $priv(t) = user$
 $connect(t, s, client_pe_auth)$

Edge Preconditions

$var(t, client_pe_auth)$
 $priv(s) \geq user$
 $priv(t) = root$
 $\neg edge(s, t, client_r2r_auth)$
 $connect(t, s, client_pe_auth)$

Postconditions

$priv(t) = root$

Exploit Based Examples

client-r2u: Client based attack that gives user privileges to an attacker with no privileges.

Main Preconditions

$var(t, client_pe_user)$
 $priv(s) \geq user$
 $priv(t) = none$
 $connect(t, s, client_pe_user)$

Edge Preconditions

$var(t, client_pe_user)$
 $priv(s) \geq user$
 $s \neq t$
 $priv(t) = user$
 $\neg edge(s, t, client_r2u)$
 $connect(t, s, client_pe_user)$

Postconditions

$priv(t) = user$

Exploit Based Examples

client-browser-trojan

client-info-system-noauth: Client based attack that leaks system information to an attacker with no privileges.

Main Preconditions

$var(t, client_leak_noauth)$
 $\neg var(t, info_system)$
 $priv(s) \geq user$
 $priv(t) < root$
 $connect(t, s, client_leak_noauth)$

Edge Preconditions

$var(t, info_system)$
 $var(t, client_leak_noauth)$
 $priv(s) \geq user$
 $priv(t) < root$
 $\neg edge(s, t, client_info_system_noauth)$
 $connect(t, s, client_leak_noauth)$

Postconditions

$var(t, info_system)$

Exploit Based Examples

client-info-system-auth: Client based attack that leaks system information to an attacker with user privilege.

Main Preconditions

$var(t, client_leak_auth)$
 $\neg var(t, info_system)$
 $priv(s) \geq user$
 $priv(t) = user$
 $connect(t, s, client_leak_auth)$

Edge Preconditions

$var(t, client_leak_auth)$
 $var(t, info_system)$
 $priv(s) \geq user$
 $priv(t) = user$
 $\neg edge(s, t, client_info_system_auth)$
 $connect(t, s, client_leak_auth)$

Postconditions

$var(t, info_system)$

Exploit Based Examples

client-info-network-noauth: Client based attack that leaks network information to an attacker with no privileges.

Main Preconditions

$var(t, client_leak_network_noauth)$
 $\neg var(info_network)$
 $priv(s) \geq user$
 $priv(t) < root$
 $connect(t, s, client_leak_network_noauth)$

Edge Preconditions

$var(info_network)$
 $var(t, client_leak_network_noauth)$
 $priv(s) \geq user$
 $priv(t) < root$
 $\neg edge(s, t, client_info_network_noauth)$
 $connect(t, s, client_leak_network_noauth)$

Postconditions

$var(info_network)$

Exploit Based Examples

client-info-network-auth: Client based attack that leaks network information to an attacker with user privilege.

Main Preconditions

$var(t, client_leak_network_auth)$
 $\neg var(info_network)$
 $priv(s) \geq user$
 $priv(t) = user$
 $connect(t, s, client_leak_network_auth)$

Edge Preconditions

$var(info_network)$
 $var(t, client_leak_network_auth)$
 $priv(s) \geq user$
 $priv(t) = user$
 $\neg edge(s, t, client_info_network_auth)$
 $connect(t, s, client_leak_network_auth)$

Postconditions

$var(info_network)$

Exploit Based Examples

router-wan-to-admin: Router attack that gives an attacker root privileges on the router and disables all firewall rules for that router.

Main Preconditions

$var(t, wan_admin)$
 $priv(t) < root$
 $priv(s) \geq user$
 $connect(s, t, wan_admin)$

Edge Preconditions

$var(t, wan_admin)$
 $priv(t) = root$
 $priv(s) \geq user$
 $\neg edge(s, t, router_wan_to_admin)$
 $connect(s, t, wan_admin)$

Postconditions

$priv(t) = root$
 $remove_firewall(t)$

Exploit Based Examples

router-firewall-disable: Router attack that disables all firewall rules for the router.

Main Preconditions

$var(t, firewall_disable)$
 $priv(s) \geq user$
 $\neg edge(s, t, firewall_disable)$
 $connect(s, t, firewall_disable)$

Edge Preconditions

Not applicable to this rule

Postconditions

$remove_firewall(t)$

Exploit Based Examples

Bibliography

- [1] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing, “Automated Generation and Analysis of Attack Graphs,” in *Proceedings IEEE Symposium on Security and Privacy*, May 2002, pp. 254 – 265.
- [2] O. Sheyner, “Scenario Graphs and Attack Graphs,” Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, April 2004.
- [3] S. Jha, O. Sheyner, and J. Wing, “Two Formal Analyses of Attack Graphs,” in *IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, June 2002, pp. 49–63.
- [4] C. L. Forgy, “Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem,” *Artificial Intelligence*, vol. 19, pp. 17–37, 1982.
- [5] E. Friedman-Hill, *JESS in Action*. Manning Publications Company, 2003.
- [6] R. Bisbey and D. Hollingworth, “Protection Analysis: Final Report,” Information Sciences Institute, University of Southern California: Marina Del Rey, CA, USA, Technical Report ISI/SR-78-13, May 1978.
- [7] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, “A Taxonomy of Computer Program Security Flaws,” *ACM Computing Surveys*, vol. 26, no. 3, pp. 211 – 254, September 1994.
- [8] U. Lindqvist and E. Jonsson, “How to Systematically Classify Computer Security Intrusions,” in *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1997, pp. 154 – 163.
- [9] S. Whalen, M. Bishop, and S. Engle, “Protocol Vulnerability Analysis,” Department of Computer Science, University of California, Davis, USA, Technical Report CSE-2005-04, 2005, draft.
- [10] S. Axelsson, “Intrusion detection systems: A survey and taxonomy,” Chalmers University, Tech. Rep. 99 – 15, Mar. 2000.
- [11] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner, “State of the practice of intrusion detection,” Carnegie Mellon Software Engineering Institute, Tech. Rep. CMU/SEI-99-TR-028, Jan. 2000.
- [12] C. Ko, G. Fink, and K. Levitt, “Automated detection of vulnerabilities in privileged programs by execution monitoring,” in *Proc. of the 10th Annual Computer Security Applications Conference*, Orlando, FL, Dec. 1994, pp. 134 – 144.

- [13] C. Ko, M. Ruschitzka, and K. Levitt, "Execution monitoring of security-critical programs in distributed systems: A specification-based approach," in *Proc. of the 1997 IEEE Symposium on Security and Privacy*, 1997, pp. 175 – 187.
- [14] P. Uppuluri and R. Sekar, "Experiences with specification-based intrusion detection," in *Proc. Recent Advances in Intrusion Detection (RAID 2001)*, Davis, CA, USA, Oct. 2001, pp. 172 – 189.
- [15] P. D. Williams, K. P. Anchor, J. L. Bebo, G. H. Gunsch, and G. D. Lamont, "CDIS: Towards a computer immune system for detecting network intrusions," in *Proc. Recent Advances in Intrusion Detection (RAID 2001)*, Davis, CA, USA, Oct. 2001, pp. 117 – 133.
- [16] J. Balthrop, F. Esponda, S. Forrest, and M. Glickman, "Coverage and generalization in an artificial immune system," in *Proc. Genetic and Evolutionary Computation Conference (GECCO 2002)*, July 2002, pp. 3 – 10.
- [17] A. Somayaji, S. Hofmeyer, and S. Forrest, "Principles of a computer immune system," in *Proc. New Security Paradigms Workshop (NSPW-97)*, Langdale, UK, 1997, pp. 75 – 82.
- [18] S. Hofmeyer and S. Forrest, "Architecture for an artificial immune system," *Evolutionary Computation Journal*, vol. 8, no. 4, pp. 433 – 473, 2000.
- [19] S. A. Hofmeyer and S. Forrest, "Immunity by design: An artificial immune system," in *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, San Francisco, CA, USA, 1999, pp. 1289 – 1296.
- [20] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri, "Self-nonsel self discrimination in a computer," in *Proc. 1994 IEEE Symposium on Research in Security and Privacy*, Los Alamitos, CA, USA, 1994, pp. 202 – 214.
- [21] S. J. Templeton and K. Levitt, "A Require/Provides Model for Computer Attacks," in *Proceedings New Security Paradigms Workshop*, Cork Island, September 2000.
- [22] R. W. Ritchey and P. Ammann, "Using Model Checking to Analyze Network Vulnerabilities," in *Proceedings, 2000 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000, pp. 156 – 165.
- [23] P. Ammann, D. Wijesekara, and S. Kaushik, "Scalable, Graph-Based Network Vulnerability Analysis," in *Proceedings CCS02: 9th ACM Conference on Computer and Communication Security*. Washington, DC: ACM, November 2002, pp. 217 – 224.
- [24] C. Phillips and L. Swiler, "A Graph-Based System for Network-Vulnerability Analysis," in *Proceedings of the New Security Paradigms Workshop*, Charlottesville, VA, 1998.
- [25] L. Swiler, C. Phillips, D. Ellis, and S. Chakerian, "Computer-Attack Graph Generation Tool," in *Proceedings of DARPA Information Survivability Conference and Exposition II*, June 2001.
- [26] S. Jajodia, S. Noel, and B. O'Berry, *Managing Cyber Threats: Issues, Approaches and Challenges*. Kluwer Academic Publisher, 2003, ch. Topological Analysis of Network Attack Vulnerability.

- [27] S. Noel and S. Jajodia, “Managing Attack Graph Complexity Through Visual Hierarchical Aggregation,” in *1st International Workshop on Visualization and Data Mining for Computer Security*, Washington, DC, USA, October 2004, pp. 109 – 118.
- [28] S. Noel, S. Jajodia, B. O’Berry, and M. Jacobs, “Efficient Minimum-Cost Network Hardening Via Exploit Dependency Graphs,” in *Proceedings of the 19th Annual Computer Security Applications Conference*, Las Vegas, NV, USA, December 2003.
- [29] K. M. C. Tan, K. S. Killourhy, and R. A. Maxion, “Undermining an anomaly-based intrusion detection system using common exploits,” in *Proc. Recent Advances in Intrusion Detection (RAID 2002)*, Zurich, Oct. 2002, pp. 54 – 73.
- [30] C. Ko, P. Brutch, J. Rowe, G. Tsafnat, and K. Levitt, “System health and intrusion monitoring using a hierarchy of constraints,” in *Proc. Recent Advances in Intrusion Detection (RAID 2001)*, Davis, CA, USA, Oct. 2001, pp. 190 – 203.
- [31] D. Knowles. (2003, Jan.) W32.Sobig.A@mm. Symantec Security Response. [Online]. Available: <http://securityresponse.symantec.com/avcenter/venc/data/w32.sobig.a@mm.%html>
- [32] R. Hassell and R. Perme. (2001, June) All versions of Microsoft Internet Information Services remote buffer overflow (SYSTEM level access). eEye Digital Security. [Online]. Available: <http://www.eeye.com/html/Research/Advisories/AD20010618.html>
- [33] R. Perme and M. Maiffret. (2001, July) .ida “Code Red” worm. eEye Digital Security. [Online]. Available: <http://www.eeye.com/html/Research/Advisories/AL20010717.html>
- [34] —. (2001, Aug.) CodeRedII worm analysis. eEye Digital Security. [Online]. Available: <http://www.eeye.com/html/Research/Advisories/AL20010804.html>
- [35] (2001, Sept.) CERT advisory CA-2001-26 Nimda worm. CERT Coordination Center. [Online]. Available: <http://www.cert.org/advisories/CA-2001-26.html>
- [36] (2003, Mar.) CodeRed.F. Symantec Security Response. [Online]. Available: <http://securityresponse.symantec.com/avcenter/venc/data/codered.f.html>
- [37] V. Bontchev and P. Ferrie. (2001, May) VBS/NewLove.A worm information. FRISK Software International and Command AntiVirus. [Online]. Available: <http://www.commandsoftware.com/virus/newlove.html>
- [38] C. Ng and P. Ferrie. (2000, Sept.) W95.Hybris.gen. Symantec Security Response. [Online]. Available: <http://www.symantec.com/avcenter/venc/data/w95.hybris.gen.html>
- [39] R. Cave. (2000, Dec.) W95.Hybris.Plugin. Symantec Security Response. [Online]. Available: <http://www.symantec.com/avcenter/venc/data/w95.hybris.plugin.html>
- [40] E. Kaspersky and A. Podrezov. (2001, Mar.) F-Secure virus descriptions: Magistr. F-Secure. [Online]. Available: <http://www.europe.f-secure.com/v-descs/magistr.shtml>
- [41] A. J. Vander, J. H. Sherman, and D. S. Luciano, *Human Physiology: The Mechanisms of Body Function*. New York: McGraw-Hill, Inc., 1994, ch. 20.

- [42] P. M. Lydyard, A. Whelan, and M. W. Fanger, *Instant Notes in Immunology*. New York: Springer, 2000.
- [43] L. N. de Castro and J. Timmis, *Artificial Immune Systems: A New Computational Intelligence Approach*. London: Springer, 2002.
- [44] Snort intrusion detection system. [Online]. Available: <http://www.snort.org/>
- [45] F. Gonzales, D. Dasgupta, and J. Gomez, "The effect of binary matching rules in negative selection," in *Proc. Genetic and Evolutionary Computation Conference (GECCO 2003)*, Chicago, IL, USA, July 2003, pp. 195 – 206.
- [46] J. D. Farmer, N. H. Packard, and A. S. Perelson, "The immune system, adaptation, and machine learning," in *Physica D*, 1986, pp. 22:187 – 204.
- [47] F. Gonzalez, D. Dasgupta, and R. Kozma, "Combining negative selection and classification techniques for anomaly detection," in *Proc. 2002 Congress on Evolutionary Computation (CEC 2002)*, Honolulu, HI, USA, May 2002, pp. 705 – 710.
- [48] T. M. Mitchell, *Machine Learning*. Boston, Massachusetts: WCB McGraw-Hill, 1997, ch. 2, 9.
- [49] D. Dasgupta, S. Yu, and N. S. Majumdar, "MILA - multilevel immune learning algorithm," in *Proc. Genetic and Evolutionary Computation Conference (GECCO 2003)*, Chicago, IL, USA, July 2003, pp. 183 – 194.
- [50] Zenomorph. (2001, Nov.) Fingerprinting port 80 attacks: A look into web server, and web application attack signatures. Whitepaper. [Online]. Available: <http://www.cgisecurity.com/papers/fingerprint-port80.txt>
- [51] ——. (2002, Mar.) Fingerprinting port 80 attacks: A look into web server, and web application attack signatures: Part 2. Whitepaper. [Online]. Available: <http://www.cgisecurity.com/papers/fingerprinting-2.txt>
- [52] (2002) Malevolence game webpage. [Online]. Available: <http://www.malevolence.com>
- [53] (1999) Darpa intrusion detection evaluation. MIT Lincoln Laboratory. [Online]. Available: <http://www.ll.mit.edu/IST/ideval/>
- [54] (2002) Bugtraq mailing list. [Online]. Available: <http://www.securityfocus.com/archive/1>
- [55] (2006, May) Graphviz - Open Source Graph Drawing Software. [Online]. Available: <http://www.graphviz.org/>
- [56] (2006, May) Symbolic Model Verifier (SMV). Carnegie Mellon University. [Online]. Available: <http://www.cs.cmu.edu/~modelcheck/smv.html>
- [57] A.Cimatti, E.Clarke, F.Giunchiglia, and M.Roveri. (2006, May) NuSMV. A New Symbolic Model Checker. [Online]. Available: <http://sra.itc.it/tools/nusmv>
- [58] (2006, May) CLIPS: A Tool for Building Expert Systems. [Online]. Available: <http://www.gbg.net/clips/CLIPS.html>

- [59] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb, "Security Analysis and Enhancements of Computer Operating Systems," Institute for Computer Sciences and Technology, National Bureau of Standards: Gaithersburg, MD, USA, Technical Report NBSIR 76-1041, 1976.
- [60] T. Aslam, I. Krsul, and E. H. Spafford, "Use of A Taxonomy of Security Faults," Department of Computer Science, Purdue University, West Lafayette, IN, USA, Technical Report TR-96-051, September 1996.
- [61] T. Aslam, "A Taxonomy of Security Faults in the UNIX Operating System," Master's thesis, Department of Computer Science, Purdue University, West Lafayette, IN, USA, 1995.
- [62] M. Bishop, "Vulnerabilities Analysis," in *Second International Symposium on Recent Advances in Intrusion Detection*, September 1999, pp. 125 – 136.
- [63] Nessus Vulnerability Scanner. Tenable Network Security. Plugins examined are from Nessus version 2.0.5. [Online]. Available: <http://www.nessus.org/>
- [64] F. Divina, M. Keijzer, and E. Marchiori, "A method for handling numerical attributes in GA-based inductive concept learners," in *Proc. Genetic and Evolutionary Computation Conference (GECCO 2003)*, Chicago, IL, USA, July 2003, pp. 898 – 908.
- [65] D. Dasgupta and F. Gonzales, "An immunity-based technique to characterize intrusions in computer networks," *IEEE Trans. Evol. Comput.*, vol. 6, no. 3, pp. 281 – 291, June 2002.
- [66] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.
- [67] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman, "On the formal semantics of statecharts," in *IEEE Symposium on Logic in Computer Science*, 1987, pp. 54–64.
- [68] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The StateMate Approach*. New York, NY, USA: McGraw-Hill, Inc., 1998.
- [69] (2001, September) OMG Unified Modeling Language Specification. Version 1.4, formal/01-09-67. [Online]. Available: <http://www.omg.org/cgi-bin/doc?formal/01-09-67>
- [70] J. L. Peterson, "Petri nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223–252, 1977.
- [71] P. Zave, "A distributed alternative to finite-state-machine specifications," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 10–36, 1985.